

EXHIBIT I

From: **David Boag** <dab@boagip.com>

Date: Wed, Mar 22, 2023 at 8:12 PM

Subject: True Return Systems LLC v. Compound Protocol, 22-cv-8483 (S.D.N.Y.)

To: security@compound.finance <security@compound.finance>, niraj@polychain.capital
<niraj@polychain.capital>

Dear Sir or Madam:

Per the Court's March 6, 2023 Order, attached please find the Summons and Complaint filed in this action against Compound Protocol.

Regards,



David A. Boag

BOAG LAW, PLLC

447 Broadway, Suite 2-270

New York, NY 10013

+1.212.203.6651

dab@boagip.com

--

Compound | Security

EXHIBIT 11



COMPOUND

Claim	'797 Claims	Use
Claim 1, lines 6-10	<p>creating at least one electronic parallel storage of a differences layer linked to a distributed computer ledger (DCL); the DCL contains an electronic transaction record by a time-sequenced value or a time-sequenced string;</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. PSDL (A1) 2. PSDL (A1) linked (A2) to DCL (A2,A3) 3. DCL (A2,A3) containing transaction records (A4), by 4. time sequenced value or string (A5) 	<p><u>COMPUTER BASED METHOD – (A0)</u></p> <ol style="list-style-type: none"> (1) "Compound is an algorithmic, autonomous interest rate protocol built for developers, to unlock a universe of open financial applications." – homepage website "compound.finance" (2) "In this paper we introduce a decentralized protocol which establishes money markets with algorithmically set interest rates based on supply and demand, allowing users to frictionlessly exchange the time value of Ethereum assets." - CCMP <p><u>CREATING PARALLEL STORAGE OF A DIFFERENCE LAYER – (A1)</u></p> <ol style="list-style-type: none"> (1) "Governance: Compound will begin with centralized control of the protocol (such as choosing the interest rate)" – CMMP (2) "Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest" – Compound Protocol Final (3) "Usually contracts that consume price feeds read the data from an AggregatorProxy contract, which itself reads the price from an underlying OffchainAggregator contract. – Oracle Infrastructure (4) "As both users, both assets, and prices are all contained within the Compound protocol, liquidation is frictionless and does not rely on any outside systems or order-books." - CCMP (5) "The Compound protocol currently relies on a price feed, maintained by our team, to determine each user's borrowing capacity and to measure liquidation thresholds." (8/19/19, "The

Claim	'797 Claims	Use
		<p>Open Oracle System", medium.com)</p> <p>(6) "Exchange Rate Stored - The last stored exchange rate for cTokens to underlying assets" – Compound Protocol Final</p> <p>(7) "Deviation threshold - when the off-chain price of an asset is witnessed to have moved more than x% of the previously reported price, an on-chain update is initiated." - Compound Oracle Improvement 47</p> <p>(8) "The [interest rate] demand curve is codified through governance ...governance will begin with centralized control" - CMMP</p> <p>(9) "the history of each interest rate, for each money market, is captured by an Interest Rate Index, which is calculated each time an interest rate changes..." - CMMP</p> <p>(10) "The market history service retrieves historical information about a market. You can use this API to find out the values of interest rates at a certain point in time. Its especially useful for making charts and graphs of the time-series values." – Compound API</p> <p><u>Note1</u>: a centralized computer system runs to gather and store both the price levels of assets (e.g. ETH, USDC), and changing interest rates relative to an asset.</p> <p><u>Note2</u>: processing use of prices and rates (including the determination of prices and rates) requires storage. This storage and processing is separate from the DCL as described by Compound's use of "centralized", "Comptroller", "off-chain", and post-Chainlink integration, node local storage</p> <p><u>DCL (cToken) IS ETHEREUM ERC-20 – (A2)</u></p> <p>(1) "assets supplied to a market are represented by an ERC-20 token balance ("cToken") which entitles the owner to an increasing quantity of the underlying asset" - CMMP</p>

Claim	'797 Claims	Use
		<p>(2) "Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin)and contains a transparent and publicly-inspectable ledger with a record of all transactions ..."</p> <p>– CMMP</p> <p><u>Note1:</u> created and imputed cTokens are technically consistent with their source asset, and both the cTokens and assets reside on a decentralized ledger.</p> <p><u>Note2:</u> cToken conversion quantities are linked to the stored interest rates.</p> <p><u>DCL IS DISTRIBUTED – (A3)</u></p> <p>(1) "Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest" – Compound Protocol Final</p> <p>(2) "Everyone who participates in the Ethereum network (every Ethereum node) keeps a copy of the state of this computer. Additionally, any participant can broadcast a request for this computer to perform arbitrary computation. Whenever such a request is broadcast, other participants on the network verify, validate, and carry out ("execute") the computation. This causes a state change in the EVM, which is committed and propagated throughout the entire network." - Intro to Ethereum</p> <p><u>Note1:</u> from '797 (col. 1, lines 42-48), "A distributed computer ledger (DCL) system is where all nodes are independently connected.... which is proofed for accuracy by a consensus system running on the decentralized network"</p> <p><u>DCL ELECTRONIC TRANSACTION RECORD – (A4)</u></p> <p>(1) "The Block. The block in Ethereum is the collection of relevant pieces of information (known as the block header), H, together with information corresponding to the comprised transactions, T, and a set of other block headers U that are known to have a parent equal to the present block's parent's parent (such blocks are known as ommers)" - EYP</p>

Claim	'797 Claims	Use
		<p>(2) "Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block." - EWP</p> <p><u>DCL TIME-SEQUENCED – (A5)</u></p> <p>(1) "timestamp: A scalar value equal to the reasonable output of Unix's time() at this block's inception" - EYP</p> <p>(2) "Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block." - EWP</p> <p><u>Note1</u>: the Ethereum ERC-20 blockchain uses an "incrementing nonce" (a scalar value) to sequence the recording of transactions (see Ethereum Yellow Paper)</p> <p><u>Note2</u>: Ethereum block identifiers and transaction records are immutably sequenced.</p>
Claim 1, lines 11-19	<p>accessing and storing a value through the at least one electronic parallel storage of the differences layer, the value from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential, wherein at least one differences processing engine running on a specialized computer system creates and stores parameters from a group comprised of a measurement differences and a descriptive differences;</p> <p>NOTES:</p>	<p><u>VALUE ACCESS & STORAGE THROUGH PSDL – (B1)</u></p> <p>(1) "the price of each asset is a median of prices from CoinbasePro, Bitnexus, Poloniex..." (Compound Finance website, FAQ)</p> <p>(2) "Compound protocol delegates the ability to set the value of assets to a committee which pools prices from the top 10 exchanges" – CCMF</p>

Claim	'797 Claims	Use
	<ol style="list-style-type: none"> 1. access and store a value (B1, B2), through 2. PSDL (A1, B1, B2) 3. value is time sequenced stream (B1) or descriptor (B2) 4. compute and store differences (A1, B1, B2) 	<p>(3) "The Compound protocol currently relies on a price feed, maintained by our team, to determine each user's borrowing capacity and to measure liquidation thresholds." - Open Oracle</p> <p>(4) "Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest" - Compound Protocol Final</p> <p>(5) "Usually contracts that consume price feeds read the data from an AggregatorProxy contract, which itself reads the price from an underlying OffchainAggregator contract. - Oracle Infrastructure</p> <p>(6) "This proposal will change the current oracle system from using Coinbase as the primary reporter of prices to Chainlink Price Feeds." - Compound Oracle Improvement 47</p> <p>(7) "This API reference provides information on how to interact directly with the Chainlink node - {see local node host http://localhost:6688, and key "USD" and value "28077"} (Chainlink Developers Documentation - API)</p> <p>(8) "CLIENT_NODE_URL - default http://localhost:6688" -Chainlink Developers Documentation - API</p> <p>(9) "The transaction can be found in the transaction history of Chainlink node.... The same transaction can be found in the dashboard of the external adapter in Google Cloud Function." - Bridging Blockchain</p> <p>(10) "Run Result: A Run Result is the result of executing a Job Spec or Task Spec. A Run Result is made up of a JSON blob, a Run Status, and an optional error field. Run Results are stored on Job Runs and Task Runs." (Developers Glossary</p> <p>(11) "The history of each interest rate, for each money market, is captured by an Interest Rate Index, which is calculated each time an interest rate changes..." - CMMP</p>

Claim	'797 Claims	Use
		<p>(12) "A Price Oracle maintains the current exchange rate of each supported asset...pools prices from the top 10 exchanges...used to determine a borrowing capacity and collateral requirements..." - CMMP</p> <p>(13) "Price Oracle Contract - The Compound Protocol uses prices from a smart contract called a price oracle. The Comptroller and Liquidate Borrow functions reference the prices in this oracle. Multiple oracles may exist for the different Compound markets." - Compound Protocol Final</p> <p>(14) "sumCollateral - Note: we use the stored exchange rate here, instead of calculating a new exchange rate for each collateral asset." - Compound Protocol Final</p> <p>(15) "The Open Oracle is a standard and SDK allowing reporters to sign key-value pairs (e.g. a price feed) that interested users can post to the blockchain. The system has a built-in view system that allows clients to easily share data and build aggregates (e.g. the median price from several sources)." (compound-finance/open-oracle) - Github Open Oracle</p> <p>(16) "The Open Price Feed accounts price data for the Compound protocol. The protocol's Comptroller contract uses it as a source of truth for prices. Prices are updated by Chainlink Price Feeds. The codebase is hosted on GitHub, and maintained by the community." (compound-finance/open-oracle)</p> <p>(17) If valid, the UniswapAnchoredView is updated with the asset's price. If invalid, the price data is not stored (Open Price Feed)</p> <p>(18) "UniswapAnchoredView only stores prices that are within an acceptable bound of the Uniswap time-weighted average price and are signed by a reporter. Also contains logic that upscales the posted prices into the format that Compound's Comptroller expects." (Open Price Feed)</p> <p><u>Note1:</u> prices and price feeds are time sequenced <u>Note2:</u> differences are processed and stored to maintain borrowing</p>

Claim	'797 Claims	Use
		<p>capacity and collateral requirements <u>Note3</u>: a system is calculating statistics (e.g. changes in interest rates, borrowing capacity, and median values) from streaming data</p> <p><u>VALUE DESCRIPTIVE – (B2)</u></p> <p>(1) “Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin....and contains a transparent and publically-inspectable ledger with a record of all transactions ...” – CMMP</p> <p>(2) “each cToken also has a comptroller, which are currently all set to address (“Code”) - CCMP</p> <p><u>Note</u>: each cToken is descriptively linked to its Ethereum asset and descriptively different and distinguished from other tokens on the Ethereum blockchain.</p>
<p>Claim 1, lines 20-28</p>	<p>storing the DCL containing an electronic transactions record on at least one of a distributed network of connected independent computers or a decentralized network of computers wherein the electronic transaction record is time sequenced, and a writing or an appending of the electronic transaction records is performed on the distributed network of connected independent computers or the decentralized network of computers;</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. store DCL on distributed or decentralized (C1), 2. transaction records (A4) are time sequenced (A4, A5), 3. writing or appending is performed on distributed or decentralized (C1, C2) 	<p><u>DCL DECENTRALIZED OR DISTRIBUTED – (C1)</u></p> <p>(1) “cTokens conform to the ERC-20 standard, and work just like other assets.” - cTokens</p> <p>(2) “Everyone who participates in the Ethereum network (every Ethereum node) keeps a copy of the state of this computer. Additionally, any participant can broadcast a request for this computer to perform arbitrary computation. Whenever such a request is broadcast, other participants on the network verify, validate, and carry out (“execute”) the computation. This causes a state change in the EVM, which is committed and propagated throughout the entire network.” - Intro to Ethereum</p> <p><u>DCL TRANSACTION RECORDS. WRITING/APPENDING – (C2)</u></p> <p>(1) “Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis</p>

Claim	'797 Claims	Use
Claim 1, lines 29-36	<p>storing the at least one electronic parallel storage of the differences layer on at least one of a centralized storage device controlled by the specialized computer system or a decentralized storage device controlled by the specialized computer system for increasing functionality and utility of the DCL, reducing data storage requirements, eliminating transmission of redundant data, and improving data security;</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. storing PSDL (A1) on centralized or decentralized (A1) 2. increasing functionality, utility of DCL (D1) 3. reducing data storage, redundant transmission (D2) 4. increase data security (D2) 	<p>state and sequentially applying every transaction in every block." - EWP</p> <p>see Ethereum White Paper – see the terms "transaction list", "blocks", "block number", and "adjacent blocks" (EWP) and see "writing" and "transactions are recorded into each block of the blockchain" (EWP)</p> <p><u>INCREASED FUNCTIONALITY AND UTILITY – (D1)</u></p> <ol style="list-style-type: none"> (1) "Useful Collateral – By holding or receiving a cToken, you can borrow from the Compound protocol" from website: (compound.finance/ctokens) (2) "In order to determine how much interest has accumulated, we take the current index value and compare it to the interest index at the time of the last event which was stored in the borrow balance" – Compound Protocol Final (3) "We keep a small residual of all interest that moves through the system"-CEO (CEO interview, CoinDesk, "Crypto Money Market Compound Lets HODL and Earn", 9/27/2018) (4) "sumCollateral – Note: we use the stored exchange rate here, instead of calculating a new exchange rate for each collateral asset." – Compound Protocol Final <p><u>DCL – REDUCED STORAGE, REDUNDANT DATA, DATA SECURITY – (D2)</u></p> <ol style="list-style-type: none"> (1) "...the developer of the protocol, currently controls the Ethereum address [-], which is the protocol admin. (Compound FAQ, 12/5/18) (2) "The admin account executes a transaction that eventually calls a malicious external contract (e.g., a malicious price oracle or underlying asset token). The malicious contract reentrantly calls a privileged function within the Comptroller (e.g., to set the close factor or change the price oracle). The call to adminOrInitializing() will return true, allowing the transaction to succeed." – Compound V2 Security Assessment

Claim	'797 Claims	Use
		<p>(3) "Deviation threshold - when the off-chain price of an asset is witnessed to have moved more than x% of the previously reported price, an on-chain update is initiated." - Compound Oracle Improvement 47</p> <p>(4) "Compound money markets are defined by an interest rate, applied to all borrowers uniformly which adjust over time..." - CMMP</p> <p>(5) "A Price Oracle maintains the current exchange rate of each supported asset...pools prices from the top 10 exchanges...used to determine a borrowing capacity and collateral requirements... (CMMP)</p> <p>(6) "The Open Price Feed accounts price data for the Compound protocol. The protocol's Comptroller contract uses it as a source of truth for prices. Prices are updated by Chainlink Price Feeds. The codebase is hosted on GitHub, and maintained by the community." (compound-finance/open-oracle)</p> <p>(7) "Compound will begin with centralized control of the protocol (such as choosing the interest rate)" - CMMP</p> <p>(8) "The [interest rate] demand curve is codified through governance ...governance will begin with centralized control" - CMMP</p> <p>(9) "the history of each interest rate, for each money market, is captured by an Interest Rate Index, which is calculated each time an interest rate changes..." - CMMP</p> <p>Note: selective centralization is used to control for security in interest rate settings and borrowing/lending limits. Note: centralization is used to maintain security on borrowing/lending Note: selective centralization reduces redundant overhead on blockchains</p>
Claim 1, lines 37-40	linking the electronic transaction record in the DCL to records of the at least one electronic parallel storage of the	<u>LINK - (E1)</u>

Claim	'797 Claims	Use
	<p>differences layer utilizing at least one time sequenced value, string, code, or key; and</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. link (E1) DCL transaction records (A2, A4) to PSDL records (A1, B1, B2), 2. utilizing time seq. value, string, code, or key (E1) 	<ol style="list-style-type: none"> (1) "Governance: Compound will begin with centralized control of the protocol (such as choosing the interest rate)" - CCMP (2) "Each asset supported on the Compound market has a corresponding USD-paired price feed reference contract and a validatorProxy contract (which allows for the UniswapAnchorView (UAV) oracle contract to be updated by the community as new markets are added)." - Compound Oracle Improvement 47 (3) "each cToken also has a comptroller, which are currently all set to address ("Code")" - see CCMP (4) "Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest" – Compound Protocol Final (5) "When a borrow is created, we store with it the principal amount and the interest index at that time." – Compound Protocol Final (6) each cToken has an administrator and is linked to the administrator through a distinct blockchain address ("Ethereum Code") – i.e. a string, code, and key - CCMP (7) Exchange Rate Stored - The last stored exchange rate for cTokens to underlying assets" – Compound Protocol Final (8) "Let assets(account) be the active list of assets (from storage) that a user has entered" – Compound Protocol Final (9) "A job needs to be registered on a Chainlink node using the job spec. After registration, a unique job ID is provided by the node. This is the identifier to use for the client to request for the execution to occur." - How Chainlink Works (10)"At the current release level, Job IDs are uniquely generated each time it's deployed. Therefore every data request has a pre-

Claim	'797 Claims	Use
Claim 1, lines 41-44	<p>imputing at least one measured differential with a descriptive identifier or at least one descriptive identifier to the electronic transaction record of the DCL through data storage and processing on the at least one electronic parallel storage of the differences layer.</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. impute differential or descriptor to DCL records (F1) through, 2. data storage and processing on PSDL (A1, B1, B2) 	<p>determined Chainlink node that can fulfill it. In other words, Chainlink has yet fully implemented the decentralized nature of the original design." - How Chainlink Works</p> <p>(11)"The Chainlink nodes rely on "Job Id" to recognize the required adapter to interact with and the workflow to process the data" - Bridging Blockchain</p> <p><u>DCL IMPUTING MEASURED DIFFERENTIAL OR DESCRIPTIVE ID. - (F1)</u></p> <ol style="list-style-type: none"> (1) "assets supplied to a market are represented by an ERC-20 token balance ("cToken") which entitles the owner to an increasing quantity of the underlying asset" - CMMP (2) "Deviation threshold - when the off-chain price of an asset is witnessed to have moved more than x% of the previously reported price, an on-chain update is initiated." - Compound Oracle Improvement 47 (3) "Compound money markets are defined by an interest rate, applied to all borrowers uniformly which adjust over time..." - CMMP (4) "...as the market earns interest, its cToken becomes convertible into an increasing quantity of the underlying asset" - CMMP (5) "Heartbeat threshold - If x minutes have passed without an update, a new on-chain update is initiated." - Compound Oracle Improvement 47

Bridging Blockchain – "Bridging Blockchain to the Real World using Chainlink", medium.com, Fang Gong, Apr. 29, 2019

CCMP – The Compound Money Market Protocol, Version 1.0, Feb 2019, <https://compound.finance/documents/Compound.Whitepaper.pdf>

CEO Interview – Coindesk, "Crypto Money Market Compound Lets HODL and Earn", 9/27/2018 - <https://www.coindesk.com/crypto-money-market-compound-lets-you-hodl-and-earn-interest>

Chainlink Developers Documentation – “Chainlink Node API”, <https://docs.chain.link/reference#runs>

Compound API Introduction, <https://compound.finance/docs/api>

compound-finance/open-oracle, <https://github.com/compound-finance/open-oracle>

Compound Protocol Final – Compound Protocol [Final] Version 2.1, <https://github.com/compound-finance/compound-protocol/blob/master/docs/CompoundProtocol.pdf>

Compound Oracle Improvement 47, passed, executed June 21, 2021, <https://compound.finance/governance/proposals/47>

Compound V2 Security Assessment, <https://github.com/trailofbits/publications/blob/master/reviews/compound-2.pdf>

cTokens – cTokens Introduction, <https://compound.finance/docs/ctokens>

Developers Glossary – Chainlink Developers Documentation – Glossary, <https://docs.chain.link/docs/glossary#config>

EWP – Ethereum White Paper – <https://github.com/ethereum/wiki/wiki/White-Paper>

EYP – Ethereum Yellow Paper – <https://ethereum.github.io/yellowpaper/paper.pdf>

FAQ - The Compound Money Market Protocol FAQs

Github Open Oracle - <https://github.com/compound-finance/open-oracle>

How Chainlink Works – “How Chainlink Works Under the Covers”, Kaleido, Jim Zhang, Feb. 12, 2020, <https://www.kaleido.io/blockchain-blog/how-chainlink-works-under-the-covers>

Intro to Ethereum, “Intro to Ethereum”, Ethereum.Org, Nov. 10, 2020, <https://ethereum.org/en/developers/docs/intro-to-ethereum/>

Open Oracle – “The Open Oracle System”, published by Compound in medium.com, Aug. 19, 2019

Open Price Feed – “Open Price Feed Introduction”, <https://compound.finance/docs/prices>

Oracle Infrastructure, Chainlink Proposal, <https://www.comp.xyz/t/oracle-infrastructure-chainlink-proposal/1272/55>



NON-LIMITING AND NON-EXHAUSTIVE

COMPOUND

Claim	'797 Claims	Use
Claim 7, lines 18-23	<p>a system having a memory device, the memory device further including a Random Access Memory (RAM);</p> <p>a processor connected to the memory device, the processor is configured to:</p> <p>create at least one electronic parallel storage of a differences layer linked to a distributed computer ledger (DCL), both the electronic parallel storage of the differences layer and the DCL containing a respective electronic transaction record, a time-sequenced value, or a time-sequenced string;</p> <p>NOTES:</p> <ol style="list-style-type: none"> 1. a system (A1) 2. create PSDL (A2) 3. linked (A3) to a DCL (A4, A5) 4. PSDL (transactions, or value, or string) (A1, A2) 5. DCL (transactions, or value, or string) (A6) 	<p><u>SYSTEM – (A1)</u></p> <ol style="list-style-type: none"> (1) "Compound is an algorithmic, autonomous interest rate protocol built for developers, to unlock a universe of open financial applications." – compound.finance DAO homepage website (2) "The Compound protocol currently relies on a price feed, maintained by our team, to determine each user's borrowing capacity and to measure liquidation thresholds." (8/19/19, "The Open Oracle System", medium.com) – Open Oracle (3) "Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin)....and contains a transparent and publicly-inspectable ledger with a record of all transactions ..." - CMMP (4) "Comptroller: The Compound protocol does not support specific tokens by default; instead, markets must be whitelisted. This is accomplished with an admin function..." - CMMP (5) "Governance: Compound will begin with centralized control of the protocol" - CMMP <p><u>CREATE PARALLEL STORAGE OF A DIFFERENCES LAYER (PSDL) – (A2)</u></p> <ol style="list-style-type: none"> (1) "Governance: Compound will begin with centralized control of the protocol" – CMMP (2) "As both users, both assets, and prices are all contained within the Compound protocol, liquidation is frictionless and

Claim	797 Claims	Use
		<p>does not rely on any outside systems or order-books.” - CCMP</p> <p>(3) “Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest” – Compound Protocol Final</p> <p>(4) “Usually contracts that consume price feeds read the data from an AggregatorProxy contract, which itself reads the price from an underlying OffchainAggregator contract. – Oracle Infrastructure</p> <p>(5) “A Price Oracle maintains the current exchange rate of each supported asset...pools prices from the top 10 exchanges ...used to determine a borrowing capacity and collateral requirements... - CMMP</p> <p>(6) “Exchange Rate Stored - The last stored exchange rate for cTokens to underlying assets” – Compound Protocol Final</p> <p>(7) “the history of each interest rate, for each money market, is captured by an Interest Rate Index, which is calculated each time an interest rate changes...” - CMMP</p> <p>(8) The Compound protocol currently relies on a price feed, maintained by our team, to determine each user’s borrowing capacity and to measure liquidation thresholds. (8/19/19 – Open Oracle) – Open Oracle</p> <p>(9) The Compound protocol does not support specific tokens by default; instead, markets must be whitelisted. This is accomplished with an admin function,” - CMMP</p> <p>(10)The Open Oracle is a standard and SDK allowing reporters to sign key-value pairs (e.g. a price feed) that interested users can post to the blockchain. The system has a built-in view system that allows clients to easily share data and build aggregates (e.g. the median price from several sources). (compound-finance/open-oracle) – Github Open Oracle</p>

Claim	797 Claims	Use
		<p><u>PSD LINKED TO A DISTRIBUTED COMPUTER LEDGER – (A3)</u></p> <p>(1) “Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin....and contains a transparent and publically-inspectable ledger with a record of all transactions ...” (CMMP)</p> <p>(2) “assets supplied to a market are represented by an ERC-20 token balance (“cToken”) which entitles the owner to an increasing quantity of the underlying asset” (CMMP)</p> <p><u>DCL (cToken) IS ETHEREUM ERC-20 – (A4)</u></p> <p>(1) “assets supplied to a market are represented by an ERC-20 token balance (“cToken”) which entitles the owner to an increasing quantity of the underlying asset” - CMMP</p> <p>(2) “Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin)and contains a transparent and publically-inspectable ledger with a record of all transactions ...” - CMMP</p> <p>(3) “The Block. The block in Ethereum is the collection of relevant pieces of information (known as the block header), H, together with information corresponding to the comprised transactions, T, and a set of other block headers U that are known to have a parent equal to the present block’s parent’s parent (such blocks are known as omms)” - EYP</p> <p><u>Note: created and imputed cTokens are technically consistent their source asset, and both the cTokens and assets reside on a decentralized ledger.</u></p> <p><u>DCL IS DISTRIBUTED – (A5)</u></p> <p>(1) “Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating accumulated interest” – Compound Protocol Final</p> <p>(2) “Everyone who participates in the Ethereum network (every</p>

Claim	'797 Claims	Use
		<p>Ethereum node) keeps a copy of the state of this computer. Additionally, any participant can broadcast a request for this computer to perform arbitrary computation. Whenever such a request is broadcast, other participants on the network verify, validate, and carry out ("execute") the computation. This causes a state change in the EVM, which is committed and propagated throughout the entire network." - Intro to Ethereum</p> <p>Note1: from '797 (col. 1, lines 42-48), "A distributed computer ledger (DCL) system is where all nodes are independently connectedwhich is proofed for accuracy by a consensus system running on the decentralized network"</p> <p><u>DCL ELECTRONIC TRANSACTION RECORD – (A6)</u></p> <p>(1) "Essentially, each transaction in the block must provide a valid state transition from what was the canonical state before the transaction was executed to some new state. Note that the state is not encoded in the block in any way; it is purely an abstraction to be remembered by the validating node and can only be (securely) computed for any block by starting from the genesis state and sequentially applying every transaction in every block." - EWP</p> <p>Source: See Ethereum White Paper – the terms "transaction" and "state transitions" (EWP) Source: See Ethereum Yellow Paper – "a secure decentralized generalized transaction ledger" (EYP)</p>
Claim 7, lines 24-26	<p>access a value from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential;</p> <p>NOTES</p> <ol style="list-style-type: none"> 1. access a value (B1, B2), 2. ...at least one published data stream (B1) ...at least one descriptive differential (B2) 	<p><u>ACCESS A VALUE / DATA STREAM – (B1)</u></p> <p>(1) "The Compound protocol currently relies on a price feed, maintained by our team, to determine each user's borrowing capacity and to measure liquidation thresholds." (8/19/19, "The Open Oracle System", medium.com) – Open Oracle</p> <p>(2) "A Price Oracle maintains the current exchange rate of each supported asset...pools prices from the top 10 exchanges ...used to determine a borrowing capacity and collateral</p>

Claim	'797 Claims	Use
		<p>requirements..." - CMMP</p> <p>(3) "We keep a small residual of all interest that moves through the system"-CEO (CEO interview, Coindesk, "Crypto Money Market Compound Lets HODL and Earn", 9/27/2018) – CEO Interview</p> <p>(4) "This proposal will change the current oracle system from using Coinbase as the primary reporter of prices to Chainlink Price Feeds." - Compound Oracle Improvement 47</p> <p>(5) "A Price Oracle maintains the current exchange rate of each supported asset...pools prices from the top 10 exchanges...used to determine a borrowing capacity and collateral requirements..." - CMMP</p> <p><u>ACCESS A VALUE / DESCRIPTOR – (B2)</u></p> <p>(1) "Each money market is unique to an Ethereum asset (such as Ether, an ERC-20 stablecoin....and contains a transparent and publicly-inspectable ledger with a record of all transactions ..." - CMMP</p> <p>(2) "Comptroller: The Compound protocol does not support specific tokens by default; instead, markets must be whitelisted. This is accomplished with an admin function, ..." – CMMP</p> <p>Note: each Ethereum asset is a descriptive difference from the native Ethereum with differing system operation from native Ethereum and each other.</p>
Claim 7, lines 27-31	<p>store the values from a group comprising of at least one time-sequenced electronically published data stream and at least one descriptive differential on the at least one electronic parallel storage of the differences layer;</p> <p>NOTES</p> <p>1. store values on PSDL (C1)</p>	<p><u>STORE VALUES ON THE PSDL – (C1)</u></p> <p>(1) "Compound will begin with centralized control of the protocol (such as choosing the interest rate)" (CMMP)</p> <p>(2) "Note that we calculate the exchangeRateStored for each collateral cToken using stored data, without calculating</p>

Claim	797 Claims	Use
2. ...at least one time-sequenced data stream...at least one descriptive differential (C1)		<p>accumulated interest” – Compound Protocol Final</p> <p>(3) “sumCollateral – Note: we use the stored exchange rate here, instead of calculating a new exchange rate for each collateral asset.” – Compound Protocol Final</p> <p>(4) “Usually contracts that consume price feeds read the data from an AggregatorProxy contract, which itself reads the price from an underlying OffchainAggregator contract. – Oracle Infrastructure</p> <p>(5) “This API reference provides information on how to interact directly with the Chainlink node – {see local node host http://localhost:6688, and key “USD” and value “28077”} - Chainlink Developers Documentation – API</p> <p>(6) “The transaction can be found in the transaction history of Chainlink node.... The same transaction can be found in the dashboard of the external adapter in Google Cloud Function.” - Bridging Blockchain</p> <p>(7) “CLIENT_NODE_URL – default http://localhost:6688” - Chainlink Developers Documentation – API</p> <p>(8) “the history of each interest rate, for each money market, is captured by an Interest Rate Index, which is calculated each time an interest rate changes...” - CMMIP</p> <p>(9) “the price of each asset is a median of prices from CoinbasePro, Bitnex, Poloniex...” - compound.finance FAQ</p> <p>(10) “The Compound protocol currently relies on a price feed, maintained by our team, to determine each user’s borrowing capacity and to measure liquidation thresholds.” (8/19/19, “The Open Oracle System”, medium.com) – Open Oracle</p> <p>(11) “We keep a small residual of all interest that moves through the system” -CEO (CEO interview, CoinDesk, “Crypto Money Market Compound Lets HODL and Earn”, 9/27/2018) – CEO</p>

Claim	797 Claims	Use
<p>Claim 7, lines 32-36</p>	<p>align and link a stored value record of the at least one electronic parallel storage of the differences layer to the electronic transaction record of the DCL utilizing at least one time sequenced value, string, code, or key; and</p> <p>NOTE</p> <ol style="list-style-type: none"> 1. align and link PSDL stored value to DCL (D1), 2. utilizing time esq. value, string, code, or key (D1, A6) 	<p>Interview</p> <p><u>ALIGN AND LINK – (D1)</u></p> <ol style="list-style-type: none"> (1) "Compound will begin with centralized control of the protocol (such as choosing the interest rate)" - CMMP (2) "Compound money markets are defined by an interest rate, applied to all borrowers uniformly which adjust over time..." - CMMP (3) "Each asset supported on the Compound market has a corresponding USD-paired price feed reference contract and a validatorProxy contract (which allows for the UniswapAnchorView (UAV) oracle contract to be updated by the community as new markets are added)." - Compound Oracle Improvement 47 (4) "Let assets(account) be the active list of assets (from storage) that a user has entered" – Compound Protocol Final (5) "...as the market earns interest, its cToken becomes convertible into an increasing quantity of the underlying asset" ... - CMMP (6) "...while computing interest, a function of time" - CMMP (7) "A job needs to be registered on a Chainlink node using the job spec. After registration, a unique job ID is provided by the node. This is the identifier to use for the client to request for the execution to occur." - How Chainlink Works
<p>Claim 7, lines 37-40</p>	<p>impute at least one measured differential with a descriptive identifier or at least one descriptive identifier to the electronic transaction record of the DCL.</p> <p>NOTES</p> <ol style="list-style-type: none"> 1. impute measured differential (E1) 	<p><u>IMPUTE MEASURED DIFFERENCE – (E1)</u></p> <ol style="list-style-type: none"> (1) "assets supplied to a market are represented by an ERC-20 token balance ("cToken") which entitles the owner to an increasing quantity of the underlying asset" - CMMP (2) "...as the market earns interest, its cToken becomes convertible into an increasing quantity of the underlying asset" - CMMP

Claim	'797 Claims	Use
		<p>(3) "Deviation threshold - when the off-chain price of an asset is witnessed to have moved more than x% of the previously reported price, an on-chain update is initiated." - Compound Oracle Improvement 47</p> <p>(4) "Compound money markets are defined by an interest rate, applied to all borrowers uniformly which adjust over time..." (CMMP)</p> <p>(5) "cTokens accumulates interest through their exchange rate -- over time, each cToken becomes convertible into an increasing amount of its underlying asset, even while the number of cTokens in your wallet stays the same." -- compound.finance website FAQ</p>

Bridging Blockchain -- "Bridging Blockchain to the Real World using Chainlink", medium.com, Fang Gong, Apr. 29, 2019

CCMP -- The Compound Money Market Protocol, Version 1.0, Feb 2019, <https://compound.finance/documents/Compound.Whitepaper.pdf>

CEO Interview -- Coindesk, "Crypto Money Market Compound Lets HODL and Earn", 9/27/2018 - <https://www.coindesk.com/crypto-money-market-compound-lets-you-hodl-and-earn-interest>

Chainlink Developers Documentation -- "Chainlink Node API", <https://docs.chain.link/reference#runs>

Compound API Introduction, <https://compound.finance/docs/api>

compound-finance/open-oracle, <https://github.com/compound-finance/open-oracle>

Compound Protocol Final -- Compound Protocol [Final] Version 2.1, <https://github.com/compound-finance/compound-protocol/blob/master/docs/CompoundProtocol.pdf>

Compound Oracle Improvement 47, passed, executed June 21, 2021, <https://compound.finance/governance/proposals/47>

Compound V2 Security Assessment, <https://github.com/trailofbits/publications/blob/master/reviews/compound-2.pdf>

cTokens -- cTokens Introduction, <https://compound.finance/docs/ctokens>

Developers Glossary – Chainlink Developers Documentation – Glossary, <https://docs.chain.link/docs/glossary#config>

EWP – Ethereum White Paper - <https://github.com/ethereum/wiki/wiki/White-Paper>

EYP – Ethereum Yellow Paper - <https://ethereum.github.io/yellowpaper/paper.pdf>

FAQ - The Compound Money Market Protocol FAQs

Github Open Oracle - <https://github.com/compound-finance/open-oracle>

How Chainlink Works – “How Chainlink Works Under the Covers”, Kaleido, Jim Zhang, Feb. 12, 2020, <https://www.kaleido.io/blockchain-blog/how-chainlink-works-under-the-covers>

Intro to Ethereum, “Intro to Ethereum”, Ethereum.Org, Nov. 10, 2020, <https://ethereum.org/en/developers/docs/intro-to-ethereum/>

Open Oracle – “The Open Oracle System”, published by Compound in medium.com, Aug. 19, 2019

Open Price Feed – “Open Price Feed Introduction”, <https://compound.finance/docs/prices>

Oracle Infrastructure, Chainlink Proposal, <https://www.comp.xyz/t/oracle-infrastructure-chainlink-proposal/1272/55>

EXHIBIT 10

Supplying Assets to the Compound Protocol

Quick Start Guide



Adam Bavosa

Follow



Feb 12, 2020 · 14 min read



The Compound Protocol is a series of interest rate markets running on the Ethereum blockchain. When users and applications supply an asset to the Compound Protocol, they begin earning a variable interest rate instantly. Interest accrues every Ethereum block (currently ~13 seconds), and users can withdraw their principal plus interest anytime.

Under the hood, users are contributing their assets to a large pool of liquidity (a “market”) that is available for other users to borrow, and they share in the interest that borrowers pay back to the pool.

When users supply assets, they receive cTokens from Compound in exchange. cTokens are ERC20 tokens that can be redeemed for their underlying assets at any time. As interest accrues to the assets supplied, cTokens are redeemable at an exchange rate (relative to the underlying asset) that constantly increases over time, based on the rate of interest earned by the underlying asset.

Non-technical users can interact with the Compound Protocol using an interface like Argent, Coinbase Wallet, or app.compound.finance; developers can create their own applications that interact with Compound's smart contracts.

In this guide, we're going to walk through **supplying assets via Web3.js JSON RPC and via proxy smart contracts that live on the blockchain**. These are two methods in which developers can write software to utilize the Compound Protocol.

There are examples in **JavaScript** and also **Solidity**.

Table of Contents for This Guide

- Compound Markets
- Connecting to the Ethereum Network
- Supplying on a Localhost Network
- Supplying on a Public Network
- How to Supply ETH to the protocol via Web3.js
- How to Supply a Supported ERC20 Token to the protocol via Solidity

If you are new to Ethereum, we suggest that you start by Setting up your Development Environment for Ethereum.

*All of the **code** referenced in this guide can be found in this **GitHub Repository**: Quick Start: Supplying Assets to the Compound Protocol.*

To copy the repository to your computer, run this on the command line after you've installed git:

```
git clone git@github.com:compound-developers/compound-supply-examples.git
```

Compound Markets

The Compound Protocol enables developers to build innovative products on DeFi. So far, we've seen crypto wallets equipped with savings APRs, a no-loss lottery system, an interest-earning system for donation income, and more.

The smart contracts that power the protocol are deployed to the Ethereum blockchain. This means that at the time of this guide's writing, the only types of assets that Compound can support are Ether and ERC-20 tokens.

The currently supported assets are listed here <https://compound.finance/markets>. Based on the different implementation of Ether (ETH) and ERC-20 tokens, we have to utilize two similar processes:

- The ETH supply method
- The ERC20 token supply method

Like mentioned earlier, when someone supplies an asset to the protocol, they are given **cTokens** in exchange. The method for getting **cETH** is different from the method for getting **cDAI**, **cUNI**, or any other **cToken** for an ERC-20 asset. We'll run through code examples and explanations for the two different asset supply methods.

When supplying Ether to the Compound protocol, an application can send ETH directly to the payable **mint** function in the **cEther** contract. Following that mint, **cEther** is minted for the wallet or contract that invoked the mint function. Remember that if you are calling this function from another smart contract, that contract needs a **payable** function in order to receive ETH when you redeem the **cTokens** later.

The operation is slightly different for **cERC20** tokens. In order to mint **cERC20** tokens, the invoking wallet or contract needs to first call the **approve** function on the **underlying token's contract**. All ERC20 token contracts have an **approve** function.

The approval needs to indicate that the corresponding **cToken** contract is permitted to take *up to the specified amount* from the sender address. Subsequently, when the **mint** function is invoked, the **cToken** contract retrieves the indicated amount of underlying tokens from the sender address, based on the prior approve call.

Example code for each method (JS and Solidity) is available, open source, in the **GitHub Repository** linked above.

Connecting to the Ethereum Network

You will need to use the contract address for the particular network that you're developing on; start by identifying the contract address for each network in the Docs. In this guide, we'll create a fork of Mainnet, which will run on our localhost; copy the Mainnet addresses.

If you want to use a public test net (like Ropsten, Görli, Kovan, or Rinkeby), make an Infura account at <https://infura.io/> to get your API key. If you are using your own localhost test net, or the production mainnet, we will also use Infura.

If you are not hosting your own Ethereum node to access the blockchain, make an Infura account before continuing.

For more on connecting to a public Ethereum network, see the instructions in Setting up your Development Environment for Ethereum.

Supplying to the Compound Protocol on a Localhost Network

To run an Ethereum local test net on your machine, we will fork the Main network (a.k.a Homestead or Mainnet). This means that you can interact with the production smart contracts in a test environment. **No real ETH will be used and no modifications to the production blockchain will occur.** If you haven't already, install Node.js. [Click here to install the LTS of Node.js and NPM.](#)

Let's install all of the dependencies required by the project.

```
cd compound-supply-examples/  
npm install  
npm install -g npx  
  
## or for yarn fans:  
## yarn install  
## yarn add global npx
```

This will install all of the dependencies listed in the **package.json** file, as well as a CLI tool called **npx**.

Run this command in a **second command line window** before you start running the code referenced later in this guide. The command spins up a test Ethereum blockchain on your localhost. It also seeds your localhost account with ERC20 tokens referenced at

the top of the script file. **Be sure to add your Infura project ID and Ethereum mnemonic as environment variables beforehand.**

```
## Set environment variables for the script to use

export MAINNET_PROVIDER_URL="https://mainnet.infura.io/v3/<YOUR
INFURA PROJECT ID HERE>"

export DEV_ETH_MNEMONIC="clutch captain shoe salt awake harvest setup
primary inmate ugly among become"

## Runs the Hardhat node locally
## Also seeds your first mnemonic account with test Ether and ERC20s

node ./scripts/run-localhost-fork.js
```

The script that we are running uses Hardhat to run an Ethereum node locally which has a *fork* of the history of Ethereum Mainnet. We have test Ether and test ERC-20 tokens that are seeded in the first account of the development mnemonic. Wait for the script logs to appear before continuing.

```
Running a hardhat localhost fork of mainnet at http://localhost:8545

Impersonating address on localhost...
0x5d3a536E4D6DbD6114cc1Ead35777bAB948E3643
Impersonating address on localhost...
0x35a18000230da775cac24873d00ff85bccded550
Impersonating address on localhost...
0x39AA39c021dfbaE8faC545936693aC917d5E7563
Local test account successfully seeded with DAI
Local test account successfully seeded with UNI
Local test account successfully seeded with USDC
DAI amount in first localhost account wallet: 100
UNI amount in first localhost account wallet: 10
USDC amount in first localhost account wallet: 100

Ready to test locally! To exit, hold Ctrl+C.
```

To change the types of ERC-20 assets provided to the account, uncomment the lines in the **amounts** object near the top of the **run-localhost-fork.js** script.

Supplying to Compound on a Public Network

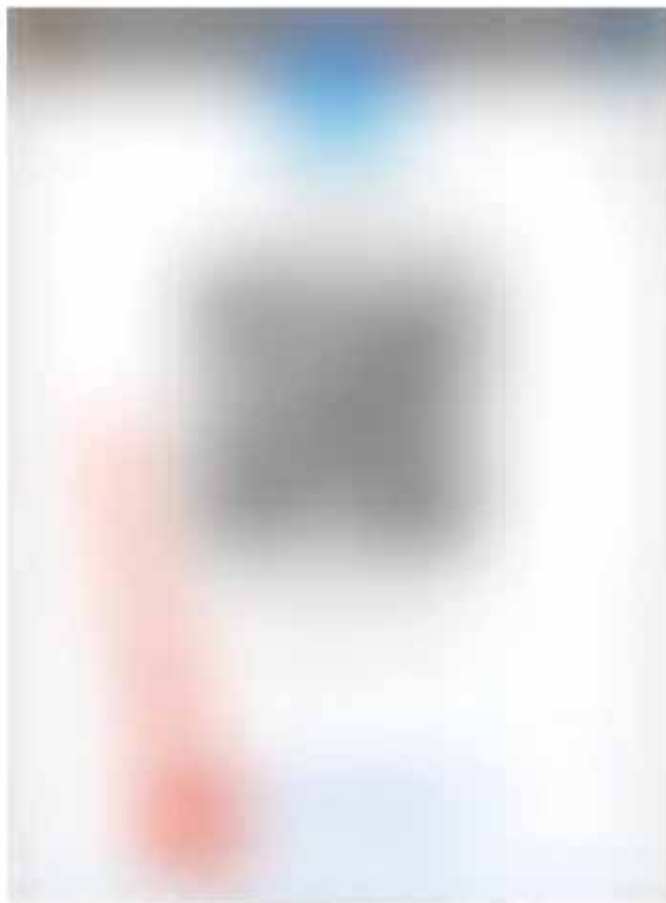
If you are supplying to the protocol on the Mainnet, Ropsten, Görli, Kovan, or Rinkeby, you should have already located and copied the **Compound contract address** for that network (see how above). You'll need it for later.

You also should have collected some ETH for that network by purchasing/mining (Main), or a test net's faucet (all the others). This is not necessary when using a localhost fork.

For example, here is Ropsten's faucet <https://faucet.ropsten.be/>. You can send yourself 1 ETH every 24 hours from a single IP address. This is test ETH that is only applicable to the Ropsten test network.

Next, copy and safely store your wallet's private key. **Don't do this if you are only testing on your localhost.** The private key is used to sign transactions that are sent on the Ethereum network. The purpose of this is to certify that the transaction was created and submitted by a unique wallet.

If you are using MetaMask for your Ethereum wallet, open the menu, click the 3 dots on the right, Account Details, Export Private Key, and input your MetaMask password. This will reveal your private key. **Keep it safe!** Copy this value and save it for later.

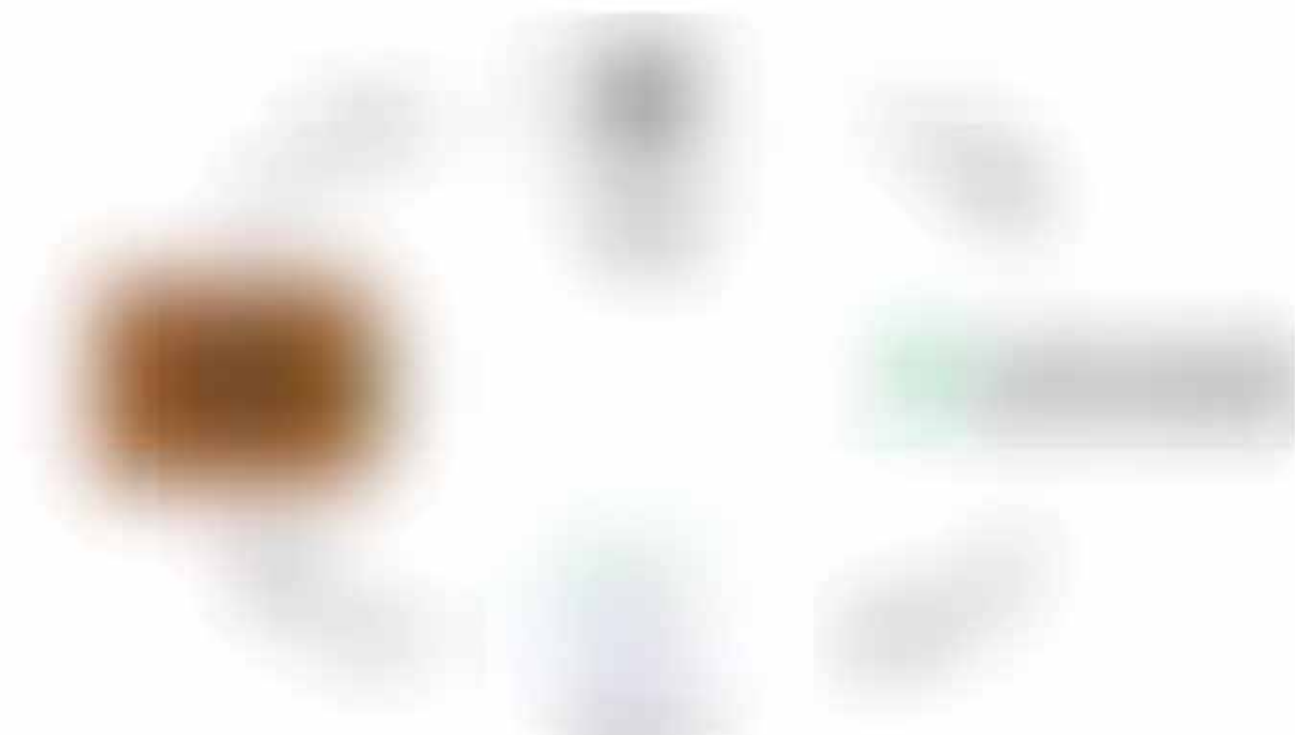




It is a **best practice** to store a test key like this as an environment variable on your local machine. When a key is stored as an environment variable, it can be referenced in code files by a variable name, instead of explicitly with a string. This promotes code cleanliness, and **reduces the risk of exposing your secret**.

Again, if you are only testing smart contracts on your localhost Hardhat node today, **don't get your MetaMask private key**. We'll rely on a private key that comes from your environment variable mnemonic (see instructions above).

How to Supply ETH to Compound via Web3.js



Supplying Ether (ETH) to the Compound Protocol is as easy as calling the “mint” function in the Compound cEther smart contract. The “mint” function transfers ETH to the Compound contract address, and mints cETH tokens. The cETH tokens are transferred to the wallet of the supplier.

Remember that the amount of ETH that can be exchanged for cETH increases every Ethereum block, which is about every 13 seconds. **There is no minimum or maximum**

amount of time that suppliers need to keep their asset in the protocol. See the varying exchange rate for each cToken by clicking on one at <https://compound.finance/markets>.

For more information on cToken concepts see the [cToken documentation](#).

In order to call the mint function, you need to first:

- Have ETH in your Ethereum wallet.
- Find your Ethereum wallet's private key.
- Connect to the network via Infura API key (see above section **Connecting to the Ethereum Network**)

There are several programming languages that have [Ethereum Web3 libraries](#), but the most popular at the time of this guide's writing is **JavaScript**.

We'll be using Node.js JavaScript to call the mint function. The following code snippets are from [this Node.js file](#) in the [supplying assets guide GitHub Repository](#). **Web browser JavaScript** is nearly identical to these code examples.

Let's import Web3.js, and initialize the Web3 object. It's pointing to our localhost's Hardhat node, which has 10000 test ETH in each of the test wallets. We get the same test wallet addresses every time we run a Hardhat node with the mnemonic environment variable (from the **Connecting to the Ethereum Network** section).

If you are using a public network (Ropsten, Kovan, etc.), make sure your wallet has ETH, and that you have your wallet private key stored as an environment variable. Also, have your Infura API key ready if you are deploying to a public test net.

Replace the HTTP provider URL in the Web3 declaration line with the appropriate network's provider if you are not using the Hardhat test environment.

Next, we'll add our wallet's private key as a variable. It's a best practice to access this as an environment variable.

If you are writing **web browser JavaScript** instead of **Node.js**, you can add the user's private key to the Web3 object by using the **ethereum.enable()** command. Here is the alternative code snippet.

Next we'll make some variables for the contract address and the contract ABI. The contract addresses are posted on this page: <https://compound.finance/docs#networks>. Remember to use the mainnet address if you are testing with Hardhat locally. The ABI is the same regardless of the Ethereum network that we are using.

The next section of code is where the magic happens. The first call in the main function supplies our ETH to the protocol by calling the **mint** function, which mints cETH. The

cETH is transferred to our wallet address.

The three subsequent function calls are not necessary, but they are here for illustration. The first method calls a getter function in the Compound contract that shows how much **underlying ETH** our cToken balance entitles us to. The second function shows our wallet's **cToken balance**. The third function gets the current **exchange rate** of cETH to ETH.

Our code sends 1 ETH to the contract, and gives our wallet cETH. The ratio of cETH to ETH should be in the ballpark of 50 to 1. Remember that the exchange rate of underlying to cToken **increases** over time.

Lastly, after the supply operation is complete, we'll redeem our cTokens. This is what a user or application will do when they want to withdraw their crypto asset from the Compound protocol.

The first method, **redeem**, redeems based on the cToken amount passed to the function call.

The second method, **redeemUnderlying**, which is commented out, redeems ETH based on the amount passed to the function call.

Finally, we execute the main function and declare an error handler.

Here is the command for running the script from the root directory of the project:

```
node ./examples-js/web3-js/supply-eth.js
```

Script example output:

How to Supply a Supported ERC20 Token to Compound via Solidity



The following will run through an example of adding an ERC20 token to the Compound protocol using Solidity smart contracts. The [full Solidity file](#) can be found in the [project GitHub repository](#).

Here's an overview of supplying a token to the Compound Protocol with Solidity:

Prerequisites

- Get some ETH into your own Ethereum wallet by purchasing/mining (or faucets on test nets). This will be used for gas costs. If you're using Hardhat on localhost, you're ready.

- Get some ERC20 token, in this case Dai. If you are working in the production environment, purchase some Dai for your Ethereum wallet. If you are working with a Hardhat test blockchain locally, your test wallet will receive some Dai when you run the **run-localhost-fork.js** script.
- Get the address of the ERC20 contract.
- Get the address of the Compound cToken contract. See **Dai** on this page: <https://compound.finance/docs#networks>.

Order of Operations

- You **transfer** Dai from your wallet to your custom contract. This is not done in Solidity, but instead with Web3.js and JSON RPC.
- You call your custom contract's function for supplying to the Compound Protocol.
- Your custom contract's function calls the approve function from the original ERC20 token contract. This allows an amount of the token to be withdrawn by cToken from your custom contract's token balance.
- Your custom contract's function calls the **mint** function in the Compound cToken contract.
- Finally, we call your custom contract's function for redeeming, to get the ERC20 token back.

Let's get started. First we'll walk through the code in our Solidity file, **MyContracts.sol**.

We added contract interfaces. The first is for our ERC20 token contract, and the second is for Compound's corresponding cToken contract.

We'll be able to call the production versions of the 3rd party contracts using these definitions. We need to initialize them with the production address of the deployed contracts, which we pass to each of the functions in **MyContract**.

The first function in **MyContract** allows the caller to supply an ERC20 token to the Compound Protocol. We will need to pass the underlying contract address, the cToken contract address, and the number of tokens we want to supply.

The function first creates references to the production instances of Dai and cDAI contracts using our interface definitions.

Then the function logs the **exchange rate** and the **supply rate**. These calls are not necessary for supplying. They are there for illustration. You can see the amounts in the “events” output later in JavaScript.

Next, our function approves the transfer of ERC20 token from our contract’s address to Compound’s cToken contract using the **approve** method.

Finally, our contract calls the cToken contract **mint** function. This supplies some Dai to the protocol, and gives our custom contract a balance of cDAI.

After we have supplied some Dai, we can redeem it at any time. The following function shows how we can accomplish that in Solidity.

The **redeemCErc20Tokens** function allows the caller to redeem based on the amount of underlying or the amount of cTokens. This is indicated by calling the function with a boolean for redeem type; True for cToken, and false for underlying amount.

If there is an error with redeeming, the error code is logged using **MyLog**. Error codes for cToken contracts are described in the documentation.

Now that we have our code written, let's run it!

Compiling

Hardhat has compilation as a simple command in the development environment. The compiler settings are configured in the **hardhat.config.js** file in the root directory of the project. Run the following command to compile the Solidity smart contracts in the **contracts** folder.

```
npx hardhat compile
```

Deploying

Once you have **deployed** your contract, the script will log the new **MyContract** address.

```
npx hardhat run ./scripts/deploy.js --network localhost

Deploying contracts with the account:
0xa0df350d2637096571f7a701cbclC5fdE30dF76A
Account balance: 10000000000000000000000
MyContract address: 0x0Bb909b7c3817F8fB7188e8fbaA2763028956E30
```

Copy this and save it for later. We'll need it to call the smart contract's function to supply Dai to Compound.

Executing

The Web3.js code that will invoke our custom smart contract can be found in the **examples-solidity/** folder. Let's run through the Web3.js **supply-erc20.js** script.

First, the script makes a Web3 object and points it to the blockchain network that we want to use to supply to Compound.

Next, we make a reference to our Ethereum wallet private key. This should be a wallet that has some ETH (for gas) and also Dai (to supply to Compound). Our script's main function first transfers Dai from our wallet to **MyContract**.

Next, we make some references to **MyContract**, the **Dai contract**, and also the **Compound cDAI contract**.

Remember, the cToken contract addresses and ABIs can be found here: <https://compound.finance/docs#networks>, and **MyContract's** address was logged when we deployed the contract.

Finally, we call our main function, which first transfers Dai from our wallet to **MyContract**.

Next we call the **supplyErc20ToCompound** function in **MyContract**, which sends 10 Dai to Compound in exchange for cDAI.

The next 2 function calls are not necessary for supplying. They illustrate how to get the **balance of underlying** ERC20 asset in the protocol and the **amount of cTokens** that

MyContract now holds.

Lastly we call the **redeemCErc20Tokens** function in **MyContract** to redeem the cDAI for Dai. The example utilizes the **redeem** method by passing a cToken amount. Under that, there is a redeem **underlying amount** example, which is commented out.

Now we're ready to run!

If you are running this on a public network, you'll need to acquire Dai for that network.

To execute the script, navigate to the project root directory and run:

```
node ./examples-solidity/web3-js/supply-erc20.js
```

If successful, the output of the script will show something like this:

Remember that this code will work with any of the ERC20 tokens that Compound supports. You will need to swap in the corresponding ERC20 token contract address and ABI into the JavaScript.

Thanks for reading! You are now able to supply assets to the Compound protocol using Solidity or JavaScript. We walked through **supplying** Ether, and also the supported ERC20 token assets. We can also **redeem** cTokens for those underlying assets later on. There are several code examples available in the Supply Examples GitHub repository.

The next key concept of developing a DApp with the Compound protocol is **borrowing assets**. Be sure to check out the next developer quick start guide: Borrowing Assets from the Compound Protocol.

Be sure to subscribe to the Compound Newsletter. Feel free to comment on this post, or get in touch in the #development room on our very active Discord server.

[Ethereum](#)[Ethereum Development](#)[Solidity Tutorial](#)[Web3](#)[Compound](#)[About](#) [Write](#) [Help](#) [Legal](#)[Get the Medium app](#)

EXHIBIT 9

COMPTROLLER



Comptroller

Introduction

The Comptroller is the risk management layer of the Compound protocol; it determines how much collateral a user is required to maintain, and whether (and by how much) a user can be liquidated. Each time a user interacts with a cToken, the Comptroller is asked to approve or deny the transaction.

The Comptroller maps user balances to prices (via the Price Oracle) to risk weights (called **Collateral Factors**) to make its determinations. Users explicitly list which assets they would like included in their risk scoring, by calling **Enter Markets** and **Exit Market**.

Architecture

The Comptroller is implemented as an upgradeable proxy. The Unitroller proxies all logic to the Comptroller implementation, but storage values are set on the Unitroller. To call Comptroller functions, use the Comptroller ABI on the Unitroller address.

Enter Markets

Enter into a list of markets - it is not an error to enter the same market more than once. In order to supply collateral or borrow in a market, it must be entered first.

Comptroller

```
function enterMarkets(address[] calldata cTokens) returns (uint[] memory)
```

- `msg.sender`: The account which shall enter the given markets.
- `cTokens`: The addresses of the cToken markets to enter.
- **RETURN**: For each market, returns an error code indicating whether or not it was entered. Each is 0 on success, otherwise an **Error code**.

Solidity

```

1 Comptroller troll = Comptroller(0xABCD...);
2 CToken[] memory cTokens = new CToken[](2);
3 cTokens[0] = CErc20(0x3FDA...);
4 cTokens[1] = CEther(0x3FDB...);
5 uint[] memory errors = troll.enterMarkets(cTokens);

```

Web31.0

```

1 const troll = Comptroller.at(0xABCD...);
2 const cTokens = [CErc20.at(0x3FDA...), CEther.at(0x3FDB...)];
3 const errors = await troll.methods.enterMarkets(cTokens).send({from: ...});

```

Exit Market

Exit a market - it is not an error to exit a market which is not currently entered. Exited markets will not count towards account liquidity calculations.

Comptroller

```

1 function exitMarket(address cToken) returns (uint)

```

- msg.sender: The account which shall exit the given market.
- cTokens: The addresses of the cToken market to exit.
- RETURN: 0 on success, otherwise an **Error code**.

Solidity

```

1 Comptroller troll = Comptroller(0xABCD...);
2 uint error = troll.exitMarket(CToken(0x3FDA...));

```

Web31.0

```
const troll = Comptroller.at(0xABCD...);
const errors = await troll.methods.exitMarket(CEther.at(0x3FDB...)).send({from: ...});
```

Get Assets In

Get the list of markets an account is currently entered into. In order to supply collateral or borrow in a market, it must be entered first. Entered markets count towards **account liquidity** calculations.

Comptroller

```
function getAssetsIn(address account) view returns (address[] memory)
```

- **account**: The account whose list of entered markets shall be queried.
- **RETURN**: The address of each market which is currently entered into.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
address[] memory markets = troll.getAssetsIn(0xMyAccount);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const markets = await troll.methods.getAssetsIn(cTokens).call();
```

Collateral Factor

A cToken's collateral factor can range from 0-90%, and represents the proportionate increase in liquidity (borrow limit) that an account receives by minting the cToken.

Generally, large or liquid assets have high collateral factors, while small or illiquid assets have low collateral factors. If an asset has a 0% collateral factor, it can't be used as collateral (or seized in

liquidation), though it can still be borrowed.

Collateral factors can be increased (or decreased) through Compound Governance, as market conditions change.

Comptroller

```
function markets(address cTokenAddress) view returns (bool, uint, bool)
```

- `cTokenAddress`: The address of the `cToken` to check if listed and get the collateral factor for.
- `RETURN`: Tuple of values (`isListed`, `collateralFactorMantissa`, `isComped`); `isListed` represents whether the comptroller recognizes this `cToken`; `collateralFactorMantissa`, scaled by `1e18`, is multiplied by a supply balance to determine how much value can be borrowed. The `isComped` boolean indicates whether or not suppliers and borrowers are distributed COMP tokens.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
(bool isListed, uint collateralFactorMantissa, bool isComped) = troll.markets(0x3FDA...);
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const result = await troll.methods.markets(0x3FDA...).call();
const {0: isListed, 1: collateralFactorMantissa, 2: isComped} = result;
```

Get Account Liquidity

Account Liquidity represents the USD value borrowable by a user, before it reaches liquidation. Users with a shortfall (negative liquidity) are subject to liquidation, and can't withdraw or borrow assets until Account Liquidity is positive again.

For each market the user has entered into, their supplied balance is multiplied by the market's collateral factor, and summed; borrow balances are then subtracted, to equal Account Liquidity. Borrowing an asset reduces Account Liquidity for each USD borrowed; withdrawing an asset reduces Account Liquidity by the asset's collateral factor times each USD withdrawn.

Because the Compound Protocol exclusively uses unsigned integers, Account Liquidity returns either a surplus or shortfall.

Comptroller

```
function getAccountLiquidity(address account) view returns (uint, uint, uint)
```

- * **account**: The account whose liquidity shall be calculated.
- * **RETURN**: Tuple of values (error, liquidity, shortfall). The error shall be 0 on success, otherwise an **error code**. A non-zero liquidity value indicates the account has available **account liquidity**. A non-zero shortfall value indicates the account is currently below his/her collateral requirement and is subject to liquidation. At most one of liquidity or shortfall shall be non-zero.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
(uint error, uint liquidity, uint shortfall) = troll.getAccountLiquidity(msg.caller);
require(error == 0, "join the Discord");
require(shortfall == 0, "account underwater");
require(liquidity > 0, "account has excess collateral");
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const result = await troll.methods.getAccountLiquidity(0xBorrower).call();
const {0: error, 1: liquidity, 2: shortfall} = result;
```

Close Factor

The percent, ranging from 0% to 100%, of a liquidatable account's borrow that can be repaid in a single liquidate transaction. If a user has multiple borrowed assets, the closeFactor applies to any single borrowed asset, not the aggregated value of a user's outstanding borrowing.

Comptroller

```
function closeFactorMantissa() view returns (uint)
```

- RETURN: The closeFactor, scaled by 1e18, is multiplied by an outstanding borrow balance to determine how much could be closed.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
uint closeFactor = troll.closeFactorMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);
const closeFactor = await troll.methods.closeFactorMantissa().call();
```

Liquidation Incentive

The additional collateral given to liquidators as an incentive to perform liquidation of underwater accounts. For example, if the liquidation incentive is 1.1, liquidators receive an extra 10% of the borrowers collateral for every unit they close.

Comptroller

```
function liquidationIncentiveMantissa() view returns (uint)
```

- RETURN: The liquidationIncentive, scaled by 1e18, is multiplied by the closed borrow amount from the liquidator to determine how much collateral can be seized.

Solidity

```
Comptroller troll = Comptroller(0xABCD...);
uint closeFactor = troll.liquidationIncentiveMantissa();
```

Web3 1.0

```
const troll = Comptroller.at(0xABCD...);  
const closeFactor = await troll.methods.liquidationIncentiveMantissa().call();
```

Key Events

Event	Description
MarketEntered(CToken cToken, address account)	Emitted upon a successful Enter Market .
MarketExited(CToken cToken, address account)	Emitted upon a successful Exit Market .

Error Codes

Code	Name	Description
0	NO_ERROR	Not a failure.
1	UNAUTHORIZED	The sender is not authorized to perform this action.
2	COMPTROLLER_MISMATCH	Liquidation cannot be performed in markets with different comptrollers.
3	INSUFFICIENT_SHORTFALL	The account does not have sufficient shortfall to perform this action.
4	INSUFFICIENT_LIQUIDITY	The account does not have sufficient liquidity to perform this action.
5	INVALID_CLOSE_FACTOR	The close factor is not valid.
6	INVALID_COLLATERAL_FACTOR	The collateral factor is not valid.

Code	Name	Description
7	INVALID_LIQUIDATION_INCENTIVE	The liquidation incentive is invalid.
8	MARKET_NOT_ENTERED	The market has not been entered by the account.
9	MARKET_NOT_LISTED	The market is not currently listed by the comptroller.
10	MARKET_ALREADY_LISTED	An admin tried to list the same market more than once.
11	MATH_ERROR	A math calculation error occurred.
12	NONZERO_BORROW_BALANCE	The action cannot be performed since the account carries a borrow balance.
13	PRICE_ERROR	The comptroller could not obtain a required price of an asset.
14	REJECTION	The comptroller rejects the action requested by the market.
15	SNAPSHOT_ERROR	The comptroller could not get the account borrows and exchange rate from the market.
16	TOO_MANY_ASSETS	Attempted to enter more markets than are currently supported
17	TOO_MUCH_REPAY	Attempted to repay more than is allowed by the protocol.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK
2	EXIT_MARKET_BALANCE_OWED
3	EXIT_MARKET_REJECTION
4	SET_CLOSE_FACTOR_OWNER_CHECK

5	SET_CLOSE_FACTOR_VALIDATION
6	SET_COLLATERAL_FACTOR_OWNER_CHECK
7	SET_COLLATERAL_FACTOR_NO_EXISTS
8	SET_COLLATERAL_FACTOR_VALIDATION
9	SET_COLLATERAL_FACTOR_WITHOUT_PRICE
10	SET_IMPLEMENTATION_OWNER_CHECK
11	SET_LIQUIDATION_INCENTIVE_OWNER_CHECK
12	SET_LIQUIDATION_INCENTIVE_VALIDATION
13	SET_MAX_ASSETS_OWNER_CHECK
14	SET_PENDING_ADMIN_OWNER_CHECK
15	SET_PENDING_IMPLEMENTATION_OWNER_CHECK
16	SET_PRICE_ORACLE_OWNER_CHECK
17	SUPPORT_MARKET_EXISTS
18	SUPPORT_MARKET_OWNER_CHECK

COMP Distribution Speeds

COMP Speed

The "COMP speed" unique to each market is an unsigned integer that specifies the amount of COMP that is distributed, per block, to suppliers and borrowers in each market. This number can be changed for individual markets by calling the `_setCompSpeed` method through a successful Compound Governance proposal.

The following is the formula for calculating the rate that COMP is distributed to each supported market.

```
utility = tokenTotalBorrows * assetPrice
utilityFraction = utility / sumOfAllCOMPedMarketUtilities
marketCompSpeed = compRate * utilityFraction
```

COMP Distributed Per Block (All Markets)

The Comptroller contract's `compRate` is an unsigned integer that indicates the rate at which the protocol distributes COMP to markets' suppliers or borrowers, every Ethereum block. The value is the amount of COMP (in wei), per block, allocated for the markets. Note that not every market has COMP distributed to its participants (see Market Metadata).

The `compRate` indicates how much COMP goes to the suppliers or borrowers, so doubling this number shows how much COMP goes to all suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to all markets.

Comptroller

```
uint public compRate;
```

Solidity

```
Comptroller troll = Comptroller(0xA8C0...);

// COMP issued per block to suppliers OR borrowers * (1 * 10 ^ 18)
uint compRate = troll.compRate();

// Approximate COMP issued per day to suppliers OR borrowers * (1 * 10 ^ 18)
uint compRatePerDay = compRate * 4 * 60 * 24;

// Approximate COMP issued per day to suppliers AND borrowers * (1 * 10 ^ 18)
uint compRatePerDayTotal = compRatePerDay * 2;
```

Web3 1.2.6

```
const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);

let compRate = await comptroller.methods.compRate().call();
compRate = compRate / 1e18;
```

```

1 // COMP issued to suppliers OR borrowers
2 const compRatePerDay = compRate * 4 * 60 * 24;
3
4 // COMP issued to suppliers AND borrowers
5 const compRatePerDayTotal = compRatePerDay * 2;

```

COMP Distributed Per Block (Single Market)

The Comptroller contract has a mapping called `compSpeeds`. It maps `cToken` addresses to an integer of each market's COMP distribution per Ethereum block. The integer indicates the rate at which the protocol distributes COMP to markets' suppliers or borrowers. The value is the amount of COMP (in wei), per block, allocated for the market. Note that not every market has COMP distributed to its participants (see Market Metadata).

The speed indicates how much COMP goes to the suppliers or the borrowers, so doubling this number shows how much COMP goes to market suppliers and borrowers combined. The code examples implement reading the amount of COMP distributed, per Ethereum block, to a single market

Comptroller

```

1 mapping(address => uint) public compSpeeds;

```

Solidity

```

1 Comptroller troll = Comptroller(0x123...);
2 address cToken = 0xabc...;
3
4 // COMP issued per block to suppliers OR borrowers * (1 * 10 ^ 18)
5 uint compSpeed = troll.compSpeeds(cToken);
6
7 // Approximate COMP issued per day to suppliers OR borrowers * (1 * 10 ^ 18)
8 uint compSpeedPerDay = compSpeed * 4 * 60 * 24;
9
10 // Approximate COMP issued per day to suppliers AND borrowers * (1 * 10 ^ 18)
11 uint compSpeedPerDayTotal = compSpeedPerDay * 2;

```

Web3 1.2.6

```

1 const cTokenAddress = '0xabc...';
2
3 const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);
4
5 let compSpeed = await comptroller.methods.compSpeeds(cTokenAddress).call();
6 compSpeed = compSpeed / 1e18;
7
8 // COMP issued to suppliers OR borrowers
9 const compSpeedPerDay = compSpeed * 4 * 60 * 24;
10
11

```

```

11 // COMP issued to suppliers AND borrowers
12 const compSpeedPerDayTotal = compSpeedPerDay * 2;

```

Claim COMP

Every Compound user accrues COMP for each block they are supplying to or borrowing from the protocol. Users may call the Comptroller's `claimComp` method at any time to transfer COMP accrued to their address.

Comptroller

```

1 // Claim all the COMP accrued by holder in all markets
2 function claimComp(address holder) public
3
4 // Claim all the COMP accrued by holder in specific markets
5 function claimComp(address holder, CToken[] memory cTokens) public
6
7 // Claim all the COMP accrued by specific holders in specific markets for their supplies and/or borrows
8 function claimComp(address[] memory holders, CToken[] memory cTokens, bool borrowers, bool suppliers) public

```

Solidity

```

1 Comptroller troll = Comptroller(0xABCD...);
2 troll.claimComp(0x1234...);

```

Web3 1.2.6

```

1 const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);
2 await comptroller.methods.claimComp("0x1234...").send({ from: sender });

```

Market Metadata

The Comptroller contract has an array called `getAllMarkets` that contains the addresses of each `cToken` contract. Each address in the `getAllMarkets` array can be used to fetch a metadata struct in the Comptroller's `markets` constant. See the [Comptroller Storage contract](#) for the Market struct definition

Comptroller

```
CToken[] public getAllMarkets;
```

Solidity

```
Comptroller troll = Comptroller(0xABCD...);  
CToken cTokens[] = troll.getAllMarkets();
```

Web3 1.2.6

```
const comptroller = new web3.eth.Contract(comptrollerAbi, comptrollerAddress);  
const cTokens = await comptroller.methods.getAllMarkets().call();  
const cToken = cTokens[0]; // address of a cToken
```

Protocol

[Markets](#)[Prices](#)[Developers](#)[Docs](#)

Governance

[Overview](#)[COMP](#)[Leaderboard](#)

Community

[Discord](#)

Forums

Grants

© 2021 Compound Labs, Inc.

EXHIBIT 8

CTOKENS



cTokens

Introduction

Each asset supported by the Compound Protocol is integrated through a cToken contract, which is an **EIP-20** compliant representation of balances supplied to the protocol. By minting cTokens, users (1) earn interest through the cToken's exchange rate, which increases in value relative to the underlying asset, and (2) gain the ability to use cTokens as collateral.

cTokens are the primary means of interacting with the Compound Protocol; when a user mints, redeems, borrows, repays a borrow, liquidates a borrow, or transfers cTokens, she will do so using the cToken contract.

There are currently two types of cTokens: CErc20 and CEther. Though both types expose the EIP-20 interface, CErc20 wraps an underlying ERC-20 asset, while CEther simply wraps Ether itself. As such, the core functions which involve transferring an asset into the protocol have slightly different interfaces depending on the type, each of which is shown below.

How do cTokens earn interest?

Each **market** has its own Supply interest rate (APR). Interest isn't distributed; instead, simply by holding cTokens, you'll earn interest.

cTokens accumulates interest through their exchange rate — over time, each cToken becomes convertible into an increasing amount of its underlying asset, even while the number of cTokens in your wallet stays the same.

Do I need to calculate the cToken exchange rate?

When a market is launched, the cToken exchange rate (how much ETH one cETH is worth) begins at 0.020000 — and increases at a rate equal to the compounding market interest rate. For example, after one year, the exchange rate might equal 0.021591.

Each user has the same cToken exchange rate; there's nothing unique to your wallet that you have to worry about.

Can you walk me through an example?

Let's say you supply 1,000 DAI to the Compound protocol, when the exchange rate is 0.020070; you would receive 49,825.61 cDAI ($1,000/0.020070$).

A few months later, you decide it's time to withdraw your DAI from the protocol; the exchange rate is now 0.021591:

- Your 49,825.61 cDAI is now equal to 1,075.78 DAI ($49,825.61 * 0.021591$)
- You could withdraw 1,075.78 DAI, which would redeem all 49,825.61 cDAI
- Or, you could withdraw a portion, such as your original 1,000 DAI, which would redeem 46,315.59 cDAI (keeping 3,510.01 cDAI in your wallet)

How do I view my cTokens?

Each cToken is visible on [Etherscan](#), and you should be able to view them in the list of tokens associated with your address

cToken balances have been integrated into [Coinbase Wallet](#) and MetaMask; other wallets may add cToken support

Can I transfer cTokens?

Yes, but exercise caution! By transferring cTokens, you're transferring your balance of the underlying asset inside the Compound protocol. If you send a cToken to your friend, your balance (viewable in the [Compound Interface](#)) will decline, and your friend will see their balance increase.

A cToken transfer will fail if the account has [entered](#) that cToken market and the transfer would have put the account into a state of negative [liquidity](#).

Mint

The mint function transfers an asset into the protocol, which begins accumulating interest based on the current **Supply Rate** for the asset. The user receives a quantity of cTokens equal to the underlying tokens supplied, divided by the current **Exchange Rate**.

CErc20

```
function mint(uint mintAmount) returns (uint)
```

- msg.sender: The account which shall supply the asset, and own the minted cTokens.
- mintAmount: The amount of the asset to be supplied, in units of the underlying asset.
- RETURN: 0 on success, otherwise an **Error code**

Before supplying an asset, users must first **approve** the cToken to access their token balance.

CEther

```
function mint() payable
```

- msg.value **payable**: The amount of ether to be supplied, in wei.
- msg.sender: The account which shall supply the ether, and own the minted cTokens.
- RETURN: No return, reverts on error.

Solidity

```
Erc20 underlying = Erc20(0xToken...); // get a handle for the underlying asset contract
CErc20 cToken = CErc20(0x3FDA...); // get a handle for the corresponding cToken contract
underlying.approve(address(cToken), 100); // approve the transfer
assert(cToken.mint(100) == 0); // mint the cTokens and assert there is no error
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
await cToken.methods.mint().send({from: myAccount, value: 50});
```

Redeem

The redeem function converts a specified quantity of cTokens into the underlying asset, and returns them to the user. The amount of underlying tokens received is equal to the quantity of cTokens redeemed, multiplied by the current **Exchange Rate**. The amount redeemed must be less than the user's **Account Liquidity** and the market's available liquidity.

CErc20 / CEther

```
function redeem(uint redeemTokens) returns (uint)
```

- msg.sender: The account to which redeemed funds shall be transferred.
- redeemTokens: The number of cTokens to be redeemed.
- RETURN: 0 on success, otherwise an **Error code**

Solidity

```
CEther cToken = CEther(0x3FDB...);
require(cToken.redeem(7) == 0, "something went wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
cToken.methods.redeem(1).send({from: ...});
```

Redeem Underlying

The redeem underlying function converts cTokens into a specified quantity of the underlying asset, and returns them to the user. The amount of cTokens redeemed is equal to the quantity of underlying tokens received, divided by the current **Exchange Rate**. The amount redeemed must be less than the user's **Account Liquidity** and the market's available liquidity.

CErc20 / CEther

```
function redeemUnderlying(uint redeemAmount) returns (uint)
```

- msg.sender: The account to which redeemed funds shall be transferred.
- redeemAmount: The amount of underlying to be redeemed.
- RETURN: 0 on success, otherwise an **Error code**

Solidity

```
CEther cToken = CEther(0x3FD0...);
require(cToken.redeemUnderlying(50) == 0, "something went wrong");
```

Web3 1.0

```
const cToken = CErc20.at(0x3FDA...);
cToken.methods.redeemUnderlying(10).send({from: ...});
```

Borrow

The borrow function transfers an asset from the protocol to the user, and creates a borrow balance which begins accumulating interest based on the **Borrow Rate** for the asset. The amount borrowed must be less than the user's **Account Liquidity** and the market's available liquidity.

To borrow Ether, the borrower must be 'payable' (solidity).

CErc20 / CEther

```
function borrow(uint borrowAmount) returns (uint)
```

- msg.sender: The account to which borrowed funds shall be transferred.
- borrowAmount: The amount of the underlying asset to be borrowed.
- RETURN: 0 on success, otherwise an **Error code**

Solidity

```
CErc20 cToken = CErc20(0x3FDA...);
require(cToken.borrow(100) == 0, "got collateral?");
```

Web31.0

```
const cToken = CEther.at(0x3F0B...);
await cToken.methods.borrow(50).send({from: 0xMyAccount});
```

Repay Borrow

The repay function transfers an asset into the protocol, reducing the user's borrow balance.

CErc20

```
function repayBorrow(uint repayAmount) returns (uint)
```

- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- `repayAmount`: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. $2^{256} - 1$) can be used to repay the full amount.
- RETURN: 0 on success, otherwise an **Error code**

Before repaying an asset, users must first **approve** the cToken to access their token balance.

CEther

```
function repayBorrow() payable
```

- `msg.value` **payable**: The amount of ether to be repaid, in wei.
- `msg.sender`: The account which borrowed the asset, and shall repay the borrow.
- RETURN: No return, reverts on error.

Solidity

```

    CToken cToken = CToken(0x3FDA...);
    require(cToken.repayBorrow.value(100)() == 0, "transfer approved?");

```

Web3 1.0

```

const cToken = CErc20.at(0x3FDA...);
cToken.methods.repayBorrow(10000).send({from: ...});

```

Repay Borrow Behalf

The repay function transfers an asset into the protocol, reducing the target user's borrow balance.

CErc20

```

function repayBorrowBehalf(address borrower, uint repayAmount) returns (uint)

```

- * msg.sender: The account which shall repay the borrow.
- * borrower: The account which borrowed the asset to be repaid.
- * repayAmount: The amount of the underlying borrowed asset to be repaid. A value of -1 (i.e. $2^{256} - 1$) can be used to repay the full amount.
- * RETURN: 0 on success, otherwise an **Error code**

Before repaying an asset, users must first **approve** the cToken to access their token balance.

CEther

```

function repayBorrowBehalf(address borrower) payable

```

- * msg.value **payable**: The amount of ether to be repaid, in wei.
- * msg.sender: The account which shall repay the borrow.
- * borrower: The account which borrowed the asset to be repaid.
- * RETURN: No return, reverts on error.

Solidity

```

CEther cToken = CEther(0x3FD8...);
require(cToken.repayBorrowBehalf.value(100)(0xBorrower) == 0, "transfer approved?");

```

Web3 1.0

```

const cToken = CErc20.at(0x3FDA...);
await cToken.methods.repayBorrowBehalf(0xBorrower, 10000).send({from: 0xPayer});

```

Transfer

Transfer is an ERC-20 method that allows accounts to send tokens to other Ethereum addresses. A cToken transfer will fail if the account has **entered** that cToken market and the transfer would have put the account into a state of negative **liquidity**.

CErc20 / CEther

```

function transfer(address recipient, uint256 amount) returns (bool)

```

- **recipient**: The transfer recipient address.
- **amount**: The amount of cTokens to transfer.
- **RETURN**: Returns a boolean value indicating whether or not the operation succeeded.

Solidity

```

CEther cToken = CEther(0x3FD8...);
cToken.transfer(0xABCD..., 100000000000);

```

Web3 1.0

```

const cToken = CErc20.at(0x3FDA...);
await cToken.methods.transfer(0xABCD..., 100000000000).send({from: 0xSender});

```

Liquidate Borrow

A user who has negative **account liquidity** is subject to **liquidation** by other users of the protocol to return his/her account liquidity back to positive (i.e. above the **collateral** requirement). When a liquidation occurs, a liquidator may repay some or all of an outstanding borrow on behalf of a borrower and in return receive a discounted amount of collateral held by the borrower; this discount is defined as the liquidation incentive.

A liquidator may close up to a certain fixed percentage (i.e. close factor) of any individual outstanding borrow of the underwater account. Unlike in v1, liquidators must interact with each cToken contract in which they wish to repay a borrow and seize another asset as collateral. When collateral is seized, the liquidator is transferred cTokens, which they may redeem the same as if they had supplied the asset themselves. Users must approve each cToken contract before calling **liquidate** (i.e. on the borrowed asset which they are repaying), as they are transferring funds into the contract.

CErc20

```
function liquidateBorrow(address borrower, uint amount, address collateral) returns (uint)
```

- **msg.sender**: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- **borrower**: The account with negative **account liquidity** that shall be liquidated.
- **repayAmount**: The amount of the borrowed asset to be repaid and converted into collateral, specified in units of the underlying borrowed asset.
- **cTokenCollateral**: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- **RETURN**: 0 on success, otherwise an **Error code**

Before supplying an asset, users must first **approve** the cToken to access their token balance.

CEther

```
function liquidateBorrow(address borrower, address cTokenCollateral) payable
```

- **msg.value** **payable**: The amount of ether to be repaid and converted into collateral, in wei.
- **msg.sender**: The account which shall liquidate the borrower by repaying their debt and seizing their collateral.
- **borrower**: The account with negative **account liquidity** that shall be liquidated.
- **cTokenCollateral**: The address of the cToken currently held as collateral by a borrower, that the liquidator shall seize.
- **RETURN**: No return, reverts on error.

Solidity

```

    CEther cToken = CEther(0x3FDB...);
    CErc20 cTokenCollateral = CErc20(0x3FDA...);
    require(cToken.liquidateBorrow.value(100)(0xBorrower, cTokenCollateral) == 0, "borrower underwater??");

```

Web3 1.0

```

const cToken = CErc20.at(0x3FDA...);
const cTokenCollateral = CEther.at(0x3FDB...);
await cToken.methods.liquidateBorrow(0xBorrower, 33, cTokenCollateral).send({from: 0xliquidator});

```

Key Events

Event	Description
Mint(address minter, uint mintAmount, uint mintTokens)	Emitted upon a successful Mint .
Redeem(address redeemer, uint redeemAmount, uint redeemTokens)	Emitted upon a successful Redeem .
Borrow(address borrower, uint borrowAmount, uint accountBorrows, uint totalBorrows)	Emitted upon a successful Borrow .
RepayBorrow(address payer, address borrower, uint repayAmount, uint accountBorrows, uint totalBorrows)	Emitted upon a successful Repay Borrow .

Event	Description
<pre>LiquidateBorrow(address liquidator, address borrower, uint repayAmount, address cTokenCollateral, uint seizeTokens)</pre>	Emitted upon a successful Liquidate Borrow .

Error Codes

Code	Name	Description
0	NO_ERROR	Not a failure.
1	UNAUTHORIZED	The sender is not authorized to perform this action.
2	BAD_INPUT	An invalid argument was supplied by the caller.
3	COMPTROLLER_REJECTION	The action would violate the comptroller policy.
4	COMPTROLLER_CALCULATION_ERROR	An internal calculation has failed in the comptroller.
5	INTEREST_RATE_MODEL_ERROR	The interest rate model returned an invalid value.
6	INVALID_ACCOUNT_PAIR	The specified combination of accounts is invalid.
7	INVALID_CLOSE_AMOUNT_REQUESTED	The amount to liquidate is invalid.
8	INVALID_COLLATERAL_FACTOR	The collateral factor is invalid.
9	MATH_ERROR	A math calculation error occurred.
10	MARKET_NOT_FRESH	Interest has not been properly accrued.
11	MARKET_NOT_LISTED	The market is not currently listed by its comptroller.

Code	Name	Description
11	MARKET_NOT_LISTED	The market is not currently listed by its comptroller.
12	TOKEN_INSUFFICIENT_ALLOWANCE	ERC-20 contract must <i>allow</i> Money Market contract to call <i>transferFrom</i> . The current <i>allowance</i> is either 0 or less than the requested supply, repayBorrow or liquidate amount.
13	TOKEN_INSUFFICIENT_BALANCE	Caller does not have sufficient balance in the ERC-20 contract to complete the desired action.
14	TOKEN_INSUFFICIENT_CASH	The market does not have a sufficient cash balance to complete the transaction. You may attempt this transaction again later.
15	TOKEN_TRANSFER_IN_FAILED	Failure in ERC-20 when transferring token into the market.
16	TOKEN_TRANSFER_OUT_FAILED	Failure in ERC-20 when transferring token out of the market.

Failure Info

Code	Name
0	ACCEPT_ADMIN_PENDING_ADMIN_CHECK
1	ACCRUE_INTEREST_ACCUMULATED_INTEREST_CALCULATION_FAILED
2	ACCRUE_INTEREST_BORROW_RATE_CALCULATION_FAILED
3	ACCRUE_INTEREST_NEW_BORROW_INDEX_CALCULATION_FAILED
4	ACCRUE_INTEREST_NEW_TOTAL_BORROWS_CALCULATION_FAILED
5	ACCRUE_INTEREST_NEW_TOTAL_RESERVES_CALCULATION_FAILED
6	ACCRUE_INTEREST_SIMPLE_INTEREST_FACTOR_CALCULATION_FAILED
7	BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED
8	BORROW_ACCRUE_INTEREST_FAILED

 BORROW_ACCRUE_INTEREST_FAILED

Code	Name
------	------

9	BORROW_CASH_NOT_AVAILABLE
---	---------------------------

10	BORROW_FRESHNESS_CHECK
----	------------------------

11	BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED
----	---

12	BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED
----	--

13	BORROW_MARKET_NOT_LISTED
----	--------------------------

14	BORROW_COMPTROLLER_REJECTION
----	------------------------------

15	LIQUIDATE_ACCRUE_BORROW_INTEREST_FAILED
----	---

16	LIQUIDATE_ACCRUE_COLLATERAL_INTEREST_FAILED
----	---

17	LIQUIDATE_COLLATERAL_FRESHNESS_CHECK
----	--------------------------------------

18	LIQUIDATE_COMPTROLLER_REJECTION
----	---------------------------------

19	LIQUIDATE_COMPTROLLER_CALCULATE_AMOUNT_SEIZE_FAILED
----	---

20	LIQUIDATE_CLOSE_AMOUNT_IS_UINT_MAX
----	------------------------------------

21	LIQUIDATE_CLOSE_AMOUNT_IS_ZERO
----	--------------------------------

22	LIQUIDATE_FRESHNESS_CHECK
----	---------------------------

23	LIQUIDATE_LIQUIDATOR_IS_BORROWER
----	----------------------------------

24	LIQUIDATE_REPAY_BORROW_FRESH_FAILED
----	-------------------------------------

25	LIQUIDATE_SEIZE_BALANCE_INCREMENT_FAILED
----	--

26	LIQUIDATE_SEIZE_BALANCE_DECREMENT_FAILED
----	--

27	LIQUIDATE_SEIZE_COMPTROLLER_REJECTION
----	---------------------------------------

28 LIQUIDATE_SEIZE_LIQUIDATOR_IS_BORROWER

29 LIQUIDATE_SEIZE_TOO_MUCH

30 MINT_ACCRUE_INTEREST_FAILED

31 MINT_COMPROLLER_REJECTION

32 MINT_EXCHANGE_CALCULATION_FAILED

33 MINT_EXCHANGE_RATE_READ_FAILED

34 MINT_FRESHNESS_CHECK

35 MINT_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED

36 MINT_NEW_TOTAL_SUPPLY_CALCULATION_FAILED

37 MINT_TRANSFER_IN_FAILED

38 MINT_TRANSFER_IN_NOT_POSSIBLE

39 REDEEM_ACCRUE_INTEREST_FAILED

40 REDEEM_COMPROLLER_REJECTION

41 REDEEM_EXCHANGE_TOKENS_CALCULATION_FAILED

42 REDEEM_EXCHANGE_AMOUNT_CALCULATION_FAILED

43 REDEEM_EXCHANGE_RATE_READ_FAILED

44 REDEEM_FRESHNESS_CHECK

45 REDEEM_NEW_ACCOUNT_BALANCE_CALCULATION_FAILED

46 REDEEM_NEW_TOTAL_SUPPLY_CALCULATION_FAILED

47 REDEEM_TRANSFER_OUT_NOT_POSSIBLE

48 REDUCE_RESERVES_ACCRUE_INTEREST_FAILED

49 REDUCE_RESERVES_ADMIN_CHECK

50 REDUCE_RESERVES_CASH_NOT_AVAILABLE

51 REDUCE_RESERVES_FRESH_CHECK

52 REDUCE_RESERVES_VALIDATION

53 REPAY_BEHALF_ACCRUE_INTEREST_FAILED

54 REPAY_BORROW_ACCRUE_INTEREST_FAILED

55 REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED

56 REPAY_BORROW_COMPROLLER_REJECTION

57 REPAY_BORROW_FRESHNESS_CHECK

58 REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED

59 REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED

60 REPAY_BORROW_TRANSFER_IN_NOT_POSSIBLE

61 SET_COLLATERAL_FACTOR_OWNER_CHECK

62 SET_COLLATERAL_FACTOR_VALIDATION

63 SET_COMPROLLER_OWNER_CHECK

64 SET_INTEREST_RATE_MODEL_ACCRUE_INTEREST_FAILED

65 SET_INTEREST_RATE_MODEL_FRESH_CHECK

66 SET_INTEREST_RATE_MODEL_OWNER_CHECK

67 SET_MAX_ASSETS_OWNER_CHECK

68 SET_ORACLE_MARKET_NOT_LISTED

69 SET_PENDING_ADMIN_OWNER_CHECK

70 SET_RESERVE_FACTOR_ACCRUE_INTEREST_FAILED

71 SET_RESERVE_FACTOR_ADMIN_CHECK

72 SET_RESERVE_FACTOR_FRESH_CHECK

73 SET_RESERVE_FACTOR_BOUNDS_CHECK

74 TRANSFER_COMPROLLER_REJECTION

75 TRANSFER_NOT_ALLOWED

76 TRANSFER_NOT_ENOUGH

77 TRANSFER_TOO_MUCH

Exchange Rate

Each cToken is convertible into an ever increasing quantity of the underlying asset, as interest accrues in the market. The exchange rate between a cToken and the underlying asset is equal to:

```
1 exchangeRate = (getCash() + totalBorrows() - totalReserves()) / totalSupply()
```

CErc20 / CEther

```
function exchangeRateCurrent() returns (uint)
```

- RETURN: The current exchange rate as an unsigned integer, scaled by $1 * 10^{(18 - 8 + \text{Underlying Token Decimals})}$.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint exchangeRateMantissa = cToken.exchangeRateCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const exchangeRate = (await cToken.methods.exchangeRateCurrent().call()) / 1e18;
```

Tip: note the use of `call` vs. `send` to invoke the function from off-chain without incurring gas costs.

Get Cash

Cash is the amount of underlying balance owned by this cToken contract. One may query the total amount of cash currently available to this market.

CErc20 / CEther

```
function getCash() returns (uint)
```

- RETURN: The quantity of underlying asset owned by the contract.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint cash = cToken.getCash();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const cash = (await cToken.methods.getCash().call());
```

Total Borrow

Total Borrows is the amount of underlying currently loaned out by the market, and the amount upon which interest is accumulated to suppliers of the market.

CErc20 / CEther

```
function totalBorrowsCurrent() returns (uint)
```

- RETURN: The total amount of borrowed underlying, with interest.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrows = cToken.totalBorrowsCurrent();
```

Web3 1.0

```
const cToken = CEther.at(0x3FDB...);
const borrows = (await cToken.methods.totalBorrowsCurrent().call());
```

Borrow Balance

A user who borrows assets from the protocol is subject to accumulated interest based on the current **borrow rate**. Interest is accumulated every block and integrations may use this function to obtain the current value of a user's borrow balance with interest.

CErc20 / CEther

```
function borrowBalanceCurrent(address account) returns (uint)
```

- account: The account which borrowed the assets.
- RETURN: The user's current borrow balance (with interest) in units of the underlying asset.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrows = cToken.borrowBalanceCurrent(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FD8...);
const borrows = await cToken.methods.borrowBalanceCurrent(account).call();
```

Borrow Rate

At any point in time one may query the contract to get the current borrow rate per block.

CErc20 / CEther

```
function borrowRatePerBlock() returns (uint)
```

- RETURN: The current borrow rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint borrowRateMantissa = cToken.borrowRatePerBlock();
```

Web3 1.0

```
const cToken = CEther.at(0x3FD8...);
const borrowRate = (await cToken.methods.borrowRatePerBlock().call()) / 1e18;
```

Total Supply

Total Supply is the number of tokens currently in circulation in this cToken market. It is part of the EIP-20 interface of the cToken contract.

CErc20 / CEther

```
function totalSupply() returns (uint)
```

- RETURN: The total number of tokens in circulation for the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint tokens = cToken.totalSupply();
```

Web3 1.0

```
const cToken = CEther.at(0x3FD8...);
const tokens = (await cToken.methods.totalSupply().call());
```

Underlying Balance

The user's underlying balance, representing their assets in the protocol, is equal to the user's cToken balance multiplied by the [Exchange Rate](#).

CErc20 / CEther

```
function balanceOfUnderlying(address account) returns (uint)
```

- **account**: The account to get the underlying balance of.
- **RETURN**: The amount of underlying currently owned by the account.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint tokens = cToken.balanceOfUnderlying(msg.caller);
```

Web3 1.0

```
const cToken = CEther.at(0x3FDA...);
const tokens = await cToken.methods.balanceOfUnderlying(account).call();
```

Supply Rate

At any point in time one may query the contract to get the current supply rate per block. The supply rate is derived from the **borrow rate**, **reserve factor** and the amount of **total borrows**.

CErc20 / CEther

```
function supplyRatePerBlock() returns (uint)
```

- **RETURN**: The current supply rate as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);
uint supplyRateMantissa = cToken.supplyRatePerBlock();
```

Web3 1.0

```
const cToken = CToken.at(0x3FD8...);  
const supplyRate = (await cToken.methods.supplyRatePerBlock().call()) / 1e18;
```

Total Reserves

Reserves are an accounting entry in each cToken contract that represents a portion of historical interest set aside as **cash** which can be withdrawn or transferred through the protocol's governance. A small portion of borrower interest accrues into the protocol, determined by the **reserve factor**.

CErc20 / CEther

```
function totalReserves() returns (uint)
```

• RETURN: The total amount of reserves held in the market.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);  
uint reserves = cToken.totalReserves();
```

Web3 1.0

```
const cToken = CToken.at(0x3FD8...);  
const reserves = (await cToken.methods.totalReserves().call());
```

Reserve Factor

The reserve factor defines the portion of borrower interest that is converted into **reserves**.

CErc20 / CEther

```
function reserveFactorMantissa() returns (uint)
```

- RETURN: The current reserve factor as an unsigned integer, scaled by 1e18.

Solidity

```
CErc20 cToken = CToken(0x3FDA...);  
uint reserveFactorMantissa = cToken.reserveFactorMantissa();
```

Web3 1.0

```
const cToken = CEther.at(0x3FD8...);  
const reserveFactor = (await cToken.methods.reserveFactorMantissa().call()) / 1e18;
```

Protocol

Markets

Prices

Developers

Docs

Governance

Overview

COMP

Leaderboard

Community

Discord

Forums

Grants

© 2021 Compound Labs, Inc.

EXHIBIT 7

compound-finance / open-oracle Public[Code](#) [Issues 1](#) [Pull requests 4](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#)

master ▾

...

open-oracle / README.md



jflatow Address audit feedback, fix up poster, and other improvements ✓

[History](#)

3 contributors



64 lines (40 sloc) | 2.01 KB

...

Open Oracle

The Open Oracle is a standard and SDK allowing reporters to sign key-value pairs (e.g. a price feed) that interested users can post to the blockchain. The system has a built-in view system that allows clients to easily share data and build aggregates (e.g. the median price from several sources).

Contracts

First, you will need solc 0.6.6 installed. Additionally for testing, you will need TypeScript installed and will need to build the open-oracle-reporter project by running `cd sdk/javascript && yarn`.

To fetch dependencies run:

```
yarn install
```

To compile everything run:

```
yarn run compile
```

To deploy contracts locally, you can run:

```
yarn run deploy --network development OpenOraclePriceData
```

Note: you will need to be running an Ethereum node locally in order for this to work. E.g., start [ganache-cli](#) in another shell.

You can add a view in `MyView.sol` and run (default is `network=development`):

```
yarn run deploy MyView arg1 arg2 ...
```

To run tests:

```
yarn run test
```

To track deployed contracts in a saddle console:

```
yarn run console
```

Reporter SDK

This repository contains a set of SDKs for reporters to easily sign "reporter" data in any supported languages. We currently support the following languages:

- [JavaScript](#) (in TypeScript)
- [Elixir](#)

Poster

The poster is a simple application that reads from a given feed (or set of feeds) and posts...

Contributing

Note: all code contributed to this repository must be licensed under each of 1. MIT, 2. BSD-3, and 3. GPLv3. By contributing code to this repository, you accept that your code is allowed to be released under any or all of these licenses or licenses in substantially similar form to these listed above.

Please submit an issue (or create a pull request) for any issues or contributions to the project. Make sure that all test cases pass, including the integration tests in the root of this project.

EXHIBIT 6

Compound FAQ



Robert Leshner

Follow



Dec 5, 2018 · 4 min read

Compound is an algorithmic, autonomous interest rate protocol— allowing users & applications to frictionlessly earn interest or borrow Ethereum assets.

If you have questions or to learn more, join the Compound [Discord](#).

Getting Started

How do I use Compound?

How to Earn Interest and Borrow Ethereum Assets

Earn interest on ETH, USDC, DAI, REP, WBTC, BAT, and ZRX

medium.com

Where else can I access the Compound protocol?

There are a number of community-built interfaces that you can use to access the Compound protocol & markets, including [Zerion](#), [InstaDapp](#), etc. You can find the full list on Compound Labs's [website](#).

How does the Compound protocol work?

Compound is the first “liquidity pool” — instead of lending assets directly to another user, you supply liquidity to a market, and users borrow from that market.

In each market, interest rates are determined algorithmically (based on supply and demand), and interest accrues every Ethereum block.

There are no pre-defined durations or terms (such as “90 days”) — you can use the Compound protocol for as short as one block, or as long as you’d like; you’re free to withdraw or repay at any time.

I’ve heard of cTokens, what are those?

When you supply assets to the Compound protocol, your balance is represented as a cToken, which can be transferred, traded, or programmed by developers to create new experiences.

Think a cToken like a receipt — it’s used to show who owns a balance inside Compound. *Please be careful — if you transfer a cToken, your balance inside Compound will decrease.*

Interest Rates

How are interest rates set?

Interest rates are a function of the liquidity available in each market, and fluctuate in real-time based on supply and demand. When liquidity is plentiful, interest rates are low. As liquidity becomes scarce, interest rates increase, incentivizing new supply and the repayment of borrowing.

You aren’t locked into an interest rate — expect it to change. On each market page, you can view the interest rate model, and graphs of interest rates (for suppliers, and borrowers) over the past two months.

Why is the supply rate lower than the borrow rate?

In each market there is excess liquidity (assets supplied > assets borrowed), which allows you to quickly withdraw or borrow funds from the protocol.

The interest paid by borrowers is earned by the suppliers of the asset. Because there are more suppliers, the interest rate they earn is proportionately lower; this measured by an asset’s *Utilization Rate*.

Second, a portion of the interest paid by borrowers is set aside as Reserves, which acts as insurance and is controlled by COMP token-holders.

On each market page, you can view the accumulated Reserves and Reserve Factor (the portion of interest set aside).

How is interest calculated?

The interest rates you see in the Interface are quoted as *annual* interest rates. Interest accrues each Ethereum block; every ~15 seconds, your balance will increase by $(1/2102400)$ of the quoted interest rate. Really!

Security

Is the Compound protocol safe? Has it been audited?

The security of the Compound protocol is our highest priority; our development team, alongside third-party auditors and consultants, has invested considerable effort to create a protocol that we believe is safe and dependable. All contract code and balances are publicly verifiable, and security researchers are eligible for a bug bounty for reporting undiscovered vulnerabilities.

Our security page contains details on each security audit, and the formal verification of the protocol.

How does the protocol's price feed work?

Compound uses the Open Price Feed, in which Reporters (like Coinbase) sign price data using a known public key, that Posters (any Ethereum address) can submit on-chain, to create a transparent, decentralized, resilient, and tamper-proof price feed.

How can I view my balance, without trusting the interface?

Sometimes, a balance appears as 0 (typically due to an issue with MetaMask or Infura). Relax — this is common.

To view your balance on the Ethereum blockchain, visit the Etherscan contract for the cToken, and scroll to `13. balanceOf`. Enter your address, click Query, and your cToken balance (with 8 decimals) will be shown.

To calculate your balance in the underlying asset, multiply your cToken balance by `4.exchangeRateStored`, and divide by `1e18`.

Help! I can't access Compound!

The Compound protocol lives on the Ethereum blockchain, and is “always-on”. In the event that MetaMask or the Compound Interface are malfunctioning, you can always access the Compound protocol manually.

Governance

Who controls the Compound protocol?

Compound is managed by a decentralized community of COMP token-holders and their delegates, who propose and vote on upgrades to the protocol.

How does Governance work?

Any address with 100 COMP can propose governance actions, which are executable code (like changes to the parameters of a market). When a proposal has gathered 100,000 COMP in support, voting begins, and lasts for 3 days. If a majority, and at least 400,000 votes are cast for the proposal, it is queued in a Timelock contract, and can be implemented after a 2 day waiting period.

Governance has complete control over the protocol, and all COMP tokens held by the protocol — the community manages the protocol as it sees fit.

How do I get involved in Compound Governance?

The community has created a Compound Forum to discuss governance proposals, and share ideas.

[About](#) [Write](#) [Help](#) [Legal](#)

Get the Medium app

